Aalto University
School of Science
Degree Programme of Computer Science and Engineering

Weiming Wu

Design and Implementation of a Shared Task Queue Groupware

Master's Thesis
Espoo, September 15, 2015

Supervisor: Docent Tomi Janhunen

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Weiming Wu

Title:
Design and Implementation of a Shared Task Queue Groupware

Date: September 15, 2015 **Pages:** 88

Professorship: Theoretical Computer Science **Code:** T-119

Supervisor: Docent Tomi Janhunen

Instructor:

Cooperation between workers in the same company or several companies has become increasingly important nowadays. The cooperation on some task usually involves sharing information about the following steps involved in the task as well as negotiation between workers who are considered to form a group. There is already software for helping people to work together and program components that can support cooperation in a particular application. Typically, they are either too specific for a certain task or too complex to configure.

In this thesis, we design groupware for handling task queues within and between companies. The groupware offers a protocol for workers in the same company to work together and to handle tasks in the shared queue. It also supports cooperation between workers in different companies. The workers cooperate in an asynchronous way but see the updates of the task queue state in real time. Information about the shared task queue is made consistent across all clients who may be physically distributed.

The thesis also compares different ways to design groupware that implements the shared task queue. A concurrency control algorithm for the application is adopted from literature and implemented. Finally, the correctness of concurrency control algorithm is assessed by developing a formal model in the Promela language and by examining the state space using the Spin model checker.

Keywords: computer supported cooperative work, distributed systems, cooperation, software, groupware, concurrency control, architecture

Language: English

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor Docent Tomi Janhunen for the continuous support of my thesis writing. His guidance helped me in the writing of this thesis and his rigorous working attitude gave me the best lesson in my working attitude.

Besides my supervisor, I would like to thank Tommi Junttila who helped a lot in my thesis writing and gave me an essential favor for working out the algorithm in this thesis.

My sincere thanks also goes to Assistant Professor Ari Serkkola for offering me the opportunities to write my thesis in his group and leading me working on diverse exciting projects.

I will never forget to thank my fellow colleague Abel Terefe, who has worked with me together in implementing our project systems and also companied me through the long period of thesis writing.

And my friends, for pointing out the lights in my future life when I feel mixed-up about my future, reminding me of pressure when I forgot moving on and giving me a hand when I got into difficulties.

Last but not least, I would like to thank my parents for trying to support me with the best they can do all these years and my girl friend for giving me the best spiritual supporting.

Espoo, September 15, 2015

Weiming Wu

Abbreviations and Acronyms

CSCW	Computer Supported Cooperative Work
HTTP	Hypertext Transfer Protocol
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
REST	Representational State Transfer
JSON	JavaScript Object Notation
OT	Operational Transformation
GOT	Generic Operational Transformation
AJAX	Asynchronous JavaScript and XML
HTML	HyperText Markup Language

Contents

Chapter 1

Introduction

By the development of information technology, computers have become involved in many aspects of our life, especially in our working life. Over the years, computers in companies have been offering various kinds of support, in document management, presentation, employee management, storing system, and so on. The use of computers has greatly eased the way people work and improved the productiveness of a single employee in a company. Last decade has witnessed a huge economical growth, partly because of the involvement of the information technology in working environments [25].

As the technologies continue to progress, software developers accumulated more and more experiences in software design and implementation. The computer software is also becoming increasingly convenient and powerful. Besides supporting the work of single employees, computer software nowadays focuses more on supporting collaborative work [32]. Such software is called collaborative software or groupware. The most popular groupware for collaborative work would be the email system, for message transmission between users [12]. Other groupware include the online shopping system, chat software, and online conference system. The most popular and successful recent groupware system might be the Google Document, a groupware used for online text editing by a group of users.

However the development of groupware is far more complicated than the building of single-user applications. The groupware would involve several users, usually distributed in different computers and in various physical locations. Each user will only know the situation of himself and conflicts between the users would be very common. A groupware is considered useless when the conflicts between users can not be solved [18, 20].

In this thesis, we aim at **presenting a software solution for a shared task queue application**. The shared task queue application is a groupware system where users can add element, i.e. tasks into the queue, remove items

from the queue, and modify the items in the queue. The queue is shared between all users and the content of the queue should be consistent when viewed by different clients. The application needs to be highly interactive, updates should be feasible in real-time and the clients should be highly distributed.

To offer a software solution for this groupware application, the thesis should address several problems. Firstly, we intend to **find a concurrency control model for our shared task queue groupware**. As it is the case in all groupware applications, concurrency control models form a necessary part in designing the application. Though traditional concurrency control methods have been researched for several decades, they might not be appropriate in the situation of groupware application. Also, although concurrency control problem has been addressed in every groupware application that has been developed, concurrency control models have to be adopted to the needs of each unique application. So, to present a software solution for our group shared task, we need have an eye on the concurrency control models in groupware applications [18].

Secondly, we target at **designing the architecture of the groupware application**. Architecture design is necessary in every software application, but architecture in groupware application is more important. It will affect the way the components in the application are deployed and also have profound effects on the user experience of the user interface. Most importantly, it almost decides how the concurrency control is going to work and how complicated it would be [15]. So, to offer a solution for the groupware application, we need to research and compare the architectures that can be applied to our application.

1.1 Structure of the Thesis

The rest of this thesis is structured as follows. In Chapter 2, we give an overview of literature essential for the groupware design, including groupware system definition, general approaches for groupware solutions, system architectures and concurrency control algorithms contained in groupware. Chapter 3 will present an introduction to the key features and the requirements for the shared task queue application. In Chapter 4, we document the architecture designed for the group task queue application and discuss the major architectural decisions that were made. Chapter 5 describes the implementation of the concurrency control algorithm and contains some discussion about the algorithm selection. Then in Chapter 6, the implementation details of the whole system are presented. The following chapter, i.e., Chapter 7 will present the evaluation in the correctness of the concurrency control

algorithm. Finally, Chapter 8 concludes the thesis.

Chapter 2

Background

Groupware applications are quite unfamiliar compared to single user applications. In this chapter, we will first introduce the concepts of a groupware system and then explore the ways developers usually choose to design how a groupware application should work. Finally an introduction to groupware architecture will be presented.

2.1 Groupware Systems

By the development of our society, the tasks people need to perform become increasingly complex. Software has long been a solution for using computers to support finishing complicated tasks, which greatly frees people from doing large amount of tedious repeated work. However, computers' powerful processors and fast memories make them so adequate in performing those repeated task that soon one witnessed a saturation in the need of more powerful processors and more advanced memories. As a result, the rapid development of computer hardware in the last decade has had little influence on the development of computer supported repeated work [19]. At the same time, the trend of work also changed greatly. Modern jobs require more social skills and cooperation. The complexity of modern jobs is featured by complex problem solving and decision making activities, rule interpretation, cooperative work processes, and so on. The increasing demands for flexibility, production time and complexity in work itself make modern jobs involve an inescapable aspect of contingency. Also, as the development of industries, the division of work is never more detailed than nowadays, which makes it necessary that large amount of people with different competence are involved in the same task. So, there is an huge requirement of cooperation in modern jobs. Moreover, the ways people choose to work are diverse. There are ideas

like adding more flexibility to working hours, making many people choose to work at home for a certain day rather than working in the company office everyday. And, due to the trend of globalization, people involved in the same task may be located in different parts of the world. As a result, the means to support cooperation are necessary. All the above described requirements call for the development of cooperative work arrangements. Cooperative work arrangements require the differentiation and combination of specialities and techniques, mutual critical assessments, and the combination of perspectives. Such arrangements include meshing, allocating and scheduling all involved actors' activities and resources, the complexity of which will increase tremendously as the number of people involved in increases. Thus, using computer for supporting cooperative work arrangement becomes necessary [10, 19, 23, 30]. These reasons stimulated the research of *Computer Supported Cooperative Work (CSCW)* for the last decade. Computer-based support for cooperative work can offer better communication facilities, which will address the geolocation problem and improve monitoring methodology and awareness mechanisms for collaborating people. Also, with techniques such as concurrency control and scheduling algorithms, the complexity of activity coordination can be greatly reduced.

2.1.1 Groupware System as a Research Field

Groupware is a type of software that is designed to support collaboration, within and between companies. The goal of such software is to help people participating in the same task work together efficiently and cooperate smoothly, regardless of various competence and geographical location of the participants. The most common experience of using groupware might be the use of e-mail, where people are exchanging information in an unstructured way [8, 19]. Other groupware packages that might be familiar to people are Lotus Notes, Microsoft Exchange, or Google Document. But what is the specific definition of groupware? Even though groupware has developed greatly in the last two decades, the definition of groupware is still in confusion and debate. The term groupware was first proposed by Peter and Trudy Jonson-Lenz, on their research notes, in October 4, 1978 and published the first definition as "intentional group processes plus software to support the group" in [24]. Later, in 1988, Peter and Trudy Jonson-Lenz gave a more detailed definition in [23]. In this definition, groupware is a generic term for specialized computerised aids that are designed for working groups. It also specified that groupware can involve software, hardware, services, and/or group process support. However, this definition excludes software that is not designed for the purpose of collaboration, such as e-mail. In 1991, Ellis

gave a broader definition as "computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment" in [16]. Even broader groupware definitions can be found in [30] and [10].

Various definitions imply that groupware may be interpreted differently by various people. Researchers have been arguing a lot about what should be considered as a groupware in [1, 17, 19], and the controversial opinions vary a lot. However, all definitions emphasise the support of collaboration. Groupware emphasizes the use of computers to facilitate human interaction [16], which implies the users of the software work together. The goal of groupware is to solve problems for working groups in communication, collaboration, and coordination, which are also the three fundamental elements of human interaction.

The research on groupware has also been considered as a problem of system design. In [19], Grudin suggests that groupware should stress the importance of "workplace democracy", though this might not be appropriate in all developing processes. So-called *Participatory Design* has been used in the design process of groupware applications, but people are more focusing on the challenges related to sociology when design a groupware application. In [22] Hughes et al. argue that "A new theoretico-empirical terrain is being formed as much in the sociology of work and organisations as elsewhere, and the interdisciplinary confrontations invoked in CSCW can be a formative influence". The sociological approaches focus on techniques for analysing the cooperation itself and emphasise a deeper understanding of the work and a new aspect of viewing the work, which is more tightly related to the cooperative software design compared to traditional systems design approaches. Schmidt and Bannon [35] even proposed as the definition of groupware that the building of information systems supporting cooperative work should focus on understanding the nature of cooperative work. This basically defined that the research field of groupware is devoted to exploring and supporting cooperative work arrangements' requirements. Thus, groupware research area is basically a design oriented research area.

The complexity of designing a groupware application is another research problem. Cooperative work is essentially distributed and various distributed factors cause a variety of concerns related to collaboration arrangements. The design process of a groupware is tightly related to the distribution of collaborative activities, i.e., time and space, the number of participants, the specialization of participants, the interdependence of different distributed parts, and the uncertainty degree of the work. The complexity of cooperation arrangements will increase as the distributed properties of cooperative work increases. The complexity grows drastically when collaborative work in-

volves multiple actors, different actors are responsible for the various aspects of the work and the interdependencies among those aspects are strong. The situation is even worse in software design, as the interdependencies among each aspect are invisible during software design [5, 7]. The traditional strategy for handling these complexities is to divide the cooperative activities into a tree like hierarchy, in which the tasks are defined and processed from top to bottom and finally the divided atomically fragmented tasks are addressed sequentially. Due to the unavoidable invisible interdependencies among those atomic fragments, multiple levels of management are required by this strategy, which will collapse in complicated working fields.

2.1.2 Requirements of a Groupware

Cooperative work involves multiple participants and the cooperation between the participants is an essential part to fulfil the task. As has been illustrated above, the cooperation arrangement task is complicated in most cooperative works. As pieces of groupware are applications used to offer computerised support to cooperation, groupware technologies offer features to solve social aspects of teamwork. The features of a groupware may have a strong effect on the team's cooperation efficiency, collaborative work result, and even the behavior or user experience of each individual inside the team. According to [15, 16, 18] several requirements of the groupware, from psychological, social, and cultural aspects, can be considered as essential parts to judge whether a groupware is successful or not.

Appropriate user interaction handling is the first requirement of groupware. Since groupware is designed to solve cooperative work problems, the interaction between users and the application is also in a form of cooperation. Different from the way that people interact with computers in single user applications, the interactive environment of groupware has certain new interaction requirements. Depending on the specific use cases, the interaction between team members can be synchronous or asynchronous. Synchronous way of interaction treats every interaction with the groupware by the team members as a sequence. The operations are handled one by one and the processed result can be seen immediately. While in asynchronized way of interaction, the interaction with the groupware by the team members may be processed at the same time. Multiple operations are supported in a way that no members of the team are blocked in waiting for processed results. Interaction in groupware is also treated in an implicit or explicit way. In an implicit way of interaction, the team members can treat the application as an single-user application and ignore all others. The interaction of other users in the same team will not affect your operations or you can ignore others'

operation. In explicit way of handling interaction, the interactions of others in the team are direct and one user's next operation is based on others' operations.

The second requirement is ***handling coordination among users in the same team***. As a solution of cooperative work, the support of cooperation in groupware mainly lies in the handling of coordinations. It can be treated as an implicit way of communication between the users in the same group. The coordination method between group members varies from application to application. The intensiveness of the coordination is strongly affected by the size of the group doing the same task. Generally speaking, simple tasks with a small group usually do not need much coordination while complicated tasks with a large group will require intensive coordination support from the groupware. Also, the way that users coordinate is strongly connected to user's experience of certain applications. For example, in Google document editing, every group member can input their own words into the text without much effect from other users. While in Google Conference, only one user's screen can be seen by all others.

Support of distribution is another requirement of groupware. Giving support to the distribution of group members is necessary in groupware. This is due to the reality that collaborative tasks need the contribution of every member in the team. For the cases that all members of the group belong to the same company, the same application needs to be run on the computer of each team member. Thus, the application needs to distribute the task to every user's computer. This implies specific requirements on techniques. For example, how to make the application run in the same state on every member's computer and address problems related to various operating system supports. For the cases that members in the same group may live in different countries and cultures, besides more strict requirements on the distributed techniques, the application must also take time zones, border crossing and social, cultural and political differences into consideration.

Visualization consistency might be the most important requirement of groupware. To avoid confusion between group members, an appropriate way of showing the application view is vital to the success of a groupware application. The main requirement here is the information consistency, thus, all the users related to the same task should get the same information. However, a strict *what-you-see-is-what-I-see* policy is not necessary, since applications run on different computers may present the information in different ways as a result of various roles in the same task [15]. Users in the same group may have various screen presentations but get identical content. In some cases, when individual views are supported, we should also present separate views to different users. All of these requirements related to visualization

will increase the implementation difficulty of the groupware.

Data hiding is the last but not least requirement. Though groupware comprises of tools used for sharing information between group members, it is also important that groupware can protect individual data. No one will try to use a groupware where all their individual data will be seen by others. Even public data that is supposed to be shared between group members, should also be invisible to members that are not in the same group. The sharing of the data should be strictly related to the rights of each user.

2.1.3 Challenges in Groupware Design and Implementation

Computer-supported cooperative work (CSCW) was first proposed in 1984 [12], and since then the interest in this field by researcher has been increasing rapidly. Algorithms related to concurrency control in groupware system have been surged up. A lot of cases that can use groupware to support collaboration have been studied. Software developers have started to implement groupware systems based on the cooperative nature of certain tasks. Many groupware products have emerged in last three decades. However, successful groupware products are relatively few and the groupware products that are popular in our life are even less frequent. This is mainly because groupware design and implementation has its own challenges compared with single-user applications. According to [7, 8, 18, 27], to build a successful groupware application, we must take additional aspects into consideration, from user requirement analysis to the final implementation of the application.

Firstly, a better understanding of task requirements is necessary [7, 34]. The task requirements include the deep insight into the cooperative work itself and an estimation of the complexity of the cooperative work. Collaborative work itself is far more complicated than single-user work, both from the structure of the work and the management of the work. The way computers support users also involves far more complexities in groupware than in single-user application, which just let the application finish repeated computational work. The collaboration related work in real life is kind of "invisible" to the user or to the application designer. Besides the tasks a groupware is designed to address, the groupware application also has to solve those invisible collaborative related tasks, such as information sharing, coordination, and so on. Hence, the designer needs to abstract all those collaborative parts from the real work circumstances into computer related work, which is far more complicated than in the case of single user application.

Secondly, an appropriate understanding of group member awareness is

required [18]. Mutual awareness means the user can know that there are others from the same group that are also involved in doing the same task. For example, in the case of Google Document, in a shared document, one can easily tell how many people are now online and editing the same document, and even find out which pieces of text are now being modified and by whom. Mutual awareness of group members is important in the use of groupware. It can help the user to have a clear view of the state of the task and also the processing stage of the task. Also, the mutual awareness in groupware is a type of social simulation that everyone's work is monitored by others and mistakes can be corrected immediately by coworkers as soon as found out. However, the presentation of mutual awareness is a challenging design task. Mutual awareness should not be treated as the main task of a groupware, since it only plays an assistant role in the whole application development. Hence, mutual awareness should not block or affect running the main task. Generally, the presentation of mutual awareness information takes place through kinds of indications, such as signals, signs, and cues. However, as the group is enlarged, the presentation of mutual awareness information can become too dominating and have severe side effects to performing the main task.

The third overall problem, similar to the development of single-user application, groupware applications also need some flexibility [3, 32]. It is common in software applications that different users for the same task may have different requirements. So, to establish a software application that can be used by all users, we have to add some flexibility to the application making it configurable to specific users. The same requirement holds in the development of groupware applications, but its realisation is more challenging. The problem here is how to build a configurable groupware application, what components a groupware application should offer and how to configure those components to add a high degree of flexibility groupware application. On the other hand, the designer should also integrate those components together and in a semantic view offer cooperation and coordination functionalities.

Other challenges include the complexity in the consideration of psychological factors, which will have effects on group members' behavior and the success of the groupware application. For example, the user of the groupware application may have no sense of the tasks' goals, lack of focus, dominate the discussion or performance in the group task, and even misunderstanding can occur. There are also difficulties in the implementation part. Since the group members can be physically distributed, the network infrastructure might cause delay problems or unreachable problems and thus, making the whole task completion meaningless.

2.2 Collaborative Application Approches

While improving the design of collaboration systems, people have long debated the approaches one should choose to build collaboration systems. Since software design showed up several decades before the idea of collaboration software, a lot of software has already been there, with appropriate designs, stable implementation architecture, mature business logic part and most importantly, a lot of users who have already been familiar with the user interfaces of those pieces of software. As a result, when designers are trying to transfer the previous single-user application to the collaboration version of the application, they need to take all those parts into consideration. Generally speaking, there are two approaches to developing collaboration systems, i.e., those based on collaboration transparency and collaboration awareness [3, 27, 30].

2.2.1 Collaboration Transparency

The collaboration transparency approach tries to inherit the whole application from a existing single-user application, but adds a collaboration module into it [3]. Designing a groupware application in this way would have a lot of benefits [3, 18, 32]. Firstly, designers can get the hints from the single-user application. Since the collaborative software is mainly the previous single user application except the collaboration part, the business logic for the software is mostly the same, which means that designers can design the application using the same architecture and same framework. These would greatly reduce the design workload and avoid failures in design. Secondly, since most part of the software would remain unchanged, it is highly possible that the developers can reuse the source code of the single-user application, which will make the development of the groupware much easier. Also, as single-user software has been successfully running for years, the reusing of source code can greatly reduce the potential for bugs in the code [29]. Thirdly, and most importantly, since a user of single-user application has long been used to the way of interacting with the single-user application user interface, the approval of users to interact with groupware as the way they interact with single user application is a great advantage. Developing groupware in this approach will enable the users to get used to groupware extremely fast. Hence, collaborative transparent groupware is more acceptable from the users' perspective. The famous groupware that was developed using the collaborative transparent approach include Google Documents. Though Google Documents do not reuse all the source code of the javascript document editor, it reuses the design of the

application and the common interface of the document editor, especially the user interfaces. Hence, at the point of being published, Google Docs was accepted by most users and has attracted more and more users over the years. The drawbacks of using collaboration transparent approach include that the design and implementation of the groupware is kind of limited by the previous single-user application. The groupware needs to be designed under the framework of the previous single-user application, even though the previous framework may not support multiple users well. Also, the collaboration transparent user interface requirement may add extra difficulties to the implementation of groupware, since some collaborations require the involvement of the user itself.

2.2.2 Collaboration Awareness

On the other hand, the idea of collaboration awareness approach is to design specific client applications for different roles in the cooperation activities [29]. This approach is building completely new applications and combining them together through the message passing protocols for the task completion. Each user is tightly limited to focusing on his own jobs, which is the essential part of collaboration in real life. According to [18, 32], there are advantages in this approach too. Firstly and obviously, this approach is trying to design a groupware based on the understanding of cooperation, which makes it completely clear for how the system is going to work, how the business logic is abstracted and each user can easily get the role related to himself based on his part in the whole task. Hence, groupware designed using this approach will make cooperation smoother and more efficient [8]. Secondly, this approach is not limited to any previous software design frameworks. Since groupware aims to solve new problems in our life using computer, previous software design experiences or frameworks could help designers to solve part of the problem but most probably will not be appropriate in the whole design. This approach gives designers more space and freedom to choose the framework that might help or even create a new framework based on the specific situation [27]. As a result, the final product will be more suitable to the specific collaborative work. Famous groupware applications that are developed using this approach would be the ordering systems for the electronic business. In such systems, the consumers only get access to the consumer views, and can choose the products they prefer, put them into the shopping cart, then choose to send the order, and pay the money. Meanwhile, the product provider is notified about the order, gets access to the provider page, checks the products they have, and then decides whether to accept this order or not. Then the confirmed order will be sent to transportation companies for deliv-

ery of the products to the customer. In this process the customer, product provider, and transportation company get totally different client pages and the cooperation between them is so tightly related to the real life of ordering products, hence the cooperation is so smooth that users might not even notice that they are cooperating with each other. Of course, this approach has its own drawbacks. The approach requires that the designer has a good understanding of the whole cooperation process and specifically designs the clients for each role, which will generally involve experts in various fields and take long developing periods.

2.3 Groupware Architectures

In analogy to single-user software, architecture also plays an important role in the design of groupware. Different from the single-user software, which targets at solving the business logic, groupware must also handle the cooperation between clients. With few users, the methods used to solve coordination between clients might be easy to come up with, and cause no problems at all. However, as the client number surges up, the coordination between clients will become a problem, as the coordination itself will involve too much information. Also, as groupware is supposed to be a distributed system, the architecture chosen for the system should take the various platform factors into consideration, otherwise the whole system may have poor compatibility with various operating systems [7]. During the past decades, groupware designers have been debating on the choice of architectures between ***centralized*** and ***replicated*** architecture.

2.3.1 Centralised Architecture

Centralized Architecture, as the name suggests, runs the main business logic in a single application and deploys it into a server or in a cloud. All the control, input, and output information is mainly handled in the central server. The clients' responsibility is only to handle the input from the user, send those pieces of information to the server, get the response information from the server, and display it on the screen [18]. According to [18, 32], the advantage of such an architecture is in the concurrency control part. As concurrency control is such a complicated problem in distributed systems, the use of centralized architecture puts all control and business logic running on one single server, which then reduces the distributed concurrency control problem into a single machine concurrency control problem. Thus, the use of centralized architecture will make the development of a groupware similar

to the development of a single user application, reducing the development period substantially. Also, the centralized architecture will reduce the efforts we need in maintaining the groupware system. As just described, the entire business logic part will run on a single server. Then the changes of the business logic will only affect the development of the central server, which is not related to various clients in different operating systems. As it is easier to develop and maintain, centralized architecture is very popular in the development of groupware. One of the most famous applications might be chess playing programs. In the game of chess, one user moves a chess piece in his client and then waits for the other player to make a move. The server is responsible for getting the input information from the user and notifying the other player.

However, there are disadvantages emerging from this architecture, in three respects: latency, bottlenecks, and compatibility. Firstly, the client must forward all input from the user to the central server, wait for the central server's replies, and then can take the other input. This means that the user after every input operation has to wait for some time to be able to provide another input. How long this period of time might be is determined by network load and the server performance. If the network is not congested, the server is powerful enough, and the business logic is simple, there will be no problem. But if any of these three factors fails, the user's waiting time will be long. Since software users are so sensitive to waiting time, the latency of centralized architecture may end up with a very bad user experience. Secondly, as all input is sent to the server and processed by the server, there is a high possibility that the server will be a performance bottleneck to the whole system. When the number of user increases, the system will be too busy in handling the coordinations and requests, thus, all clients' input will be delayed. Just as described above, this will cause bad user experience and also affect the efficiency of cooperation. Thirdly, as groupware systems are generally distributed systems, the display of the output information will be different from one client to another. Since all input and output is processed by the central server, which will treat all input and output with no difference, ignore the uniqueness of platform that the clients runs on, the client has to be specialised in order to display all pieces information locally correct. This will be an extra burden to the developers.

2.3.2 Replicated Architecture

Replicated architectures have a copy of the groupware program running on each client machine. The copies of the program can run independently of each other but coordination information is sent and received by the clients.

The input from the user is handled by the client itself and then displayed on the screen. There is no central server in the whole system. Replicated systems are designed to solve problems centralized systems suffer from. There are several benefits in using replicated architecture, according to [16, 32]. Firstly, it is low latency. Since all inputs are handled on the client machine, no network transfer is needed during the phase of handling local inputs and each client only serves for a single user, in a way that the client will only have performance problems when the task itself is causing too much calculation. Thus, it is highly possible that users do not need to wait for the clients and the cooperation will proceed smoothly. Secondly, since there is no central server in the whole system and the workload for the tasks are evenly distributed to clients, there will be no bottleneck in a specific part of the system. The less powerful computers will not have effects on the whole system, since all other clients run independently without waiting for the other clients. Thirdly, the various operating systems the clients are running on will not add extra burdens to the developers, since each client is independent and the communication protocols for different devices can be implemented platform independently. Since mature products always have a lot of users, products based on the replicated architecture are also common. For example, in Google Docs, each client page is an independent javascript document editor. The users of the Google Docs can edit the group documents independently, and the exit of any user will not affect others. Conflicting inputs of independent clients will be addressed by the concurrency control algorithm automatically. Even when the network is down, people can still input the text and synchronize the information when the network is accessible again.

The disadvantages for replicated architectures are also obvious. The coordination between different clients will be difficult to implement. There is no global sequence of operations in the whole system and each client runs in parallel, with its own notion of time. Also, the performance diverges through the whole system, which will add extra concern to the coordination algorithms. Also, the arrival time of the same operation from the same client to different devices may be different, which means that the clients need to transform the operations based on specific situations. The coordination algorithm in groupware client machine has long been a research topic. Different protocols are chosen by different applications, but they are all complicated and not easy to implement.

Nowadays, developers prefer using architectures that combine these two architectures together. Though the cost might be higher, the combined architectures are more stable, easier to implement, and less buggy.

Chapter 3

Project Requirements

This thesis targets at offering a software solution for the group shared task queue application contained in a real project. There are several requirements on the software solution. In this chapter, there will be a detailed introduction to all these requirements.

3.1 RESTful API

RESTful API is the first consideration in building web based applications, which has gained widespread acceptance. RESTful API stands for Representational State Transfer API, which typically transfers information through *Hypertext Transfer Protocol (HTTP)* with HTTP verbs that are commonly used by web browsers. The RESTful architecture style was developed by W3C Technical Architecture Group and with the use of HTTP 1.1, World Wide Web has been the largest system that was built based on the RESTful Architecture [13].

According to [4, 33], the use of RESTful Architecture will bring abundant benefits to a system. Firstly, the use of RESTful Architecture will greatly simplify the interface, since the format of all requests sent to server must follow the standard HTTP verbs. The use of the standard HTTP verbs in every request will make it clear and self explainable, hence the misuse or misunderstanding of the interface will be greatly reduced. Also, with the use of standard HTTP verbs developers can explicitly operate a piece of data in various ways, by using different HTTP verbs. Secondly, using RESTful Architecture will increase the scalability of the whole system. RESTful API requires the system to be stateless, which means that the same *uniform resource identifier (URI)* will always lead to the same data information. The state of the user is not stored in the server side, but recovered by the server

based on the request sent from the client side. Hence, a request sent by a user to different server or services will lead to the same state of the user. This can greatly improve the scalability of a system, as techniques such as load balancing can be easily adopted for redirecting requests to distribute the work load between several server machines. Also, the stateless requirement makes the modification of the server components much easier. The server only needs to take care of the request sent by the user and return the response based on the request. No other specific limitations are added. The user interface will not be affected by the change of implementation on the server side. Thirdly, the use of RESTful API will make the portability of the server or service be good. As described above, the standard HTTP verbs require identical *URI* for the same data information. Hence, the clients will always get the same response with the same request. As a result, the same service can be used by various client sides, as long as the client sides call the standard *URI*. The last but not least benefit of RESTful API is the reliability of the service. Since the service using RESTful API is easy to scale, it is also quite easy to use other server components that run a copy of the service in cases of server failure. Also, the use of standard HTTP verbs and identical *URI* will make the transfer of the request from the failed server to the working server easy and make no difference to the client users.

All the benefits described above are motivations for building our server using the RESTful API architecture, and applying all the rules required by the RESTful Architecture to our application. Also, the groupware component in our application will be used in other applications with high possibility, so enhancing the portability using RESTful Architecture is necessary in the implementation of the groupware component. Moreover, when developers take the scalability into consideration, as the number of users for the service increases, more users will be involved in the same group task. Thus, there is a potential requirement of scaling up the server in the future. Further more, since the server is a real business related application, it is supposed to be used by companies everyday. One single failure of the server might cause severe problems to the customer companies and affect the profits of the companies. Thus the application requires a high tolerance for the failure. So, due to the benefits that the RESTful Architecture can bring to the design and implementation of our application and the actual requirements of our application, designing our application with the RESTful Architecture will be our basic requirements of our application.

3.2 Offline Operation

Our application is highly distributed. In addition to the central server, there are also various client processes. Some of the client processes might run in browsers, which have very good network connections and can be online for most of the situations. Some of the client processes, however, are running on mobile phones, or iPads, which might confront the situations that the network connection is not good enough. For those mobile clients, the groupware component needs to work appropriately even without network connection.

For offline clients, the role they are playing in the cooperative task is equally important and even necessary. For them, the application should run smoothly and efficiently, especially without blocking, so that they can focus on the task and be not interrupted by the bad user experience when using the offline application. For groupware components, things get complicated if no network connection exists. The environment without networking connection means that there are no communication with the server side. As a result, coordination scheme based on the central architecture might not work here. The central architecture requires the clients to send each operation to the central server, and only after the central server has sent the response information back, can the clients continue for the next operation. Since the network is not available, using centralised architecture for coordination in groupware might result in a totally blocked application. Hence, to avoid that the application is blocked as a reason of the poor network connection, the groupware component needs to run independently of the central server. This calls for the implementation of coordination architecture in the groupware to follow the replicated architecture or similar. The replicated architecture does not rely on the central server and each client can continue to run even there is no connection with the server [16]. Also, with the use of some mechanism for storing the operation done by the users when offline, the work done by the user when offline can be uploaded to the server when the network is available [23]. In this way, the users are totally free from the network connection, and can work on the application as it is a single user application. However, as what has been pointed out in the background part, replicated architecture usually means that there will be much more challenges in implementation.

Firstly, taking the concurrency control into consideration, replicated architecture requires the concurrency control algorithm to run in a distributed way. Every client needs to consider the operations that have been done by other clients, and figure out the correct sequence of operations based on the algorithm. This is a much more complicated algorithm, compared to the concurrency control algorithm used in the central server based architecture.

Secondly, as mentioned above, some techniques are necessary to ensure that the operations done offline are correctly transferred to the server, when the network is available. According to [26] the problems when uploading the offline operations to the server, might cause conflicts with other clients, who are also operating the same data. Hence, some algorithm is needed here to transform certain local operations into some different operations, after combining with the other operations already done by others. Also, the data transferred between the clients and the server must be tolerant to failures, which will require techniques to ensure the data consistency between the client side and the server side. The technique used in the application is the message queuing, which allows the client side to store the data locally and as if the data had already been stored in the server database. This technique has been used in large scale modern distributed systems to guarantee the data transformation between distributed components.

3.3 Late Joining Users

Being a groupware application, it is not common that the all users for the same task log on the system at the same time or start the group task only when all users are on line. The real situation is that users join the task sequentially and the group task begins when the first user starts the task. Users that join after the beginning of the task are called *late joining users*. The support of *late joining users* will bring more feasibility to the groupware system and enable it to be used in more practical situations.

The support of *late joining users* basically means that let the *late joining users* take part in the group task at any time they want. After joining, the user can get all information related the the group task in a correct format, including all the data that the group task owns, the results that have been processed, and the current state of the group task, hence, the users can get involved in the group task immediately. Since users are distributed over different machines, the *late joining users* will make the system suffer from problems like passing incorrect or inconsistent information from client to client, when the joining concurrency mechanism is not well implemented.

For example, at some point, one client that has already been in the client list, operate on some piece of data on his local screen. Then this data will be updated first in his local machine, and the modifying of data will be notified to all other clients in the client list. Finally, the database in the server will be updated. Unfortunately, before the database is updated, but after the client has checked up the client list, a new client joins into the server. It is obvious that this client will not get the modification notification of the modified data,

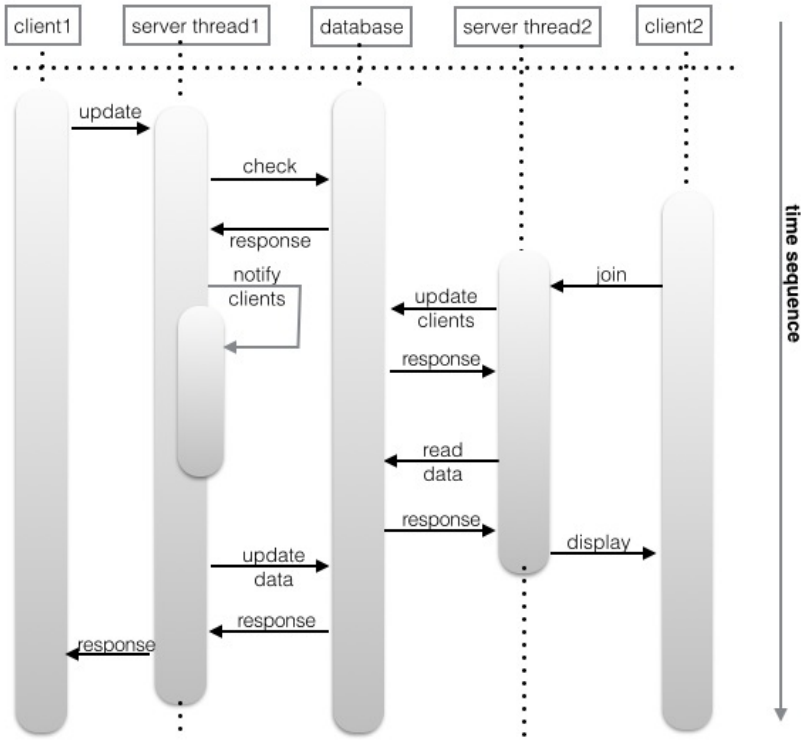

Figure 3.1: Information Inconsistency for Late Join Users

since when the updating client is checking the client list, the new user is not there. However, the new user can not get the modified data from checking the database upon joining the group task either, since the database has not been refreshed yet. So, the problem that the new client will not get all the information of the group task exists. When we change the sequence into first updating the database and then notifying the clients in the client list, there would be situations where the new joining clients might get the update notification twice, which will also lead to data inconsistency between clients. Figure 3.1 illustrates how inconsistent data could emerge.

3.4 Non-blocking IO

Input and output handling in computer systems are always the bottleneck of performance. Compared to the calculation speed of modern computers, the input and output are at so slow speed, since they are used to interact with human beings. In web applications, the input and output are basically the

user interface related layers and the database resolve layer.

According to [14, 41], the traditional way of handling *Input and Output (IO)* in the computer system is simply the system waiting for some started input or output task to complete. Since the *IO* interfaces are slower than the computer's processor, the system processes will be blocked by such slower operations and hence the resources of the system can not be taken into full use. If we use blocked *IO*, in most cases, the system will end up wasting most of the time in waiting for the *IO* to respond, thus, it will be very inefficient. This is specifically the reality in web applications, since it has so huge amount of requests from the user and so many queries into the database. Especially in the groupware of groupware, besides the normal business related request from client to server, the server also needs to handle the coordination requests from clients to server, which always exist, and as the number of users get involved into the same group task increases, the system might result in waiting for *IO* all the time, if the *IO* is handled using blocking mechanisms.

By contrast, Non-blocking *IO* handles the input and output in an asynchronous way. When *IO* information is requested the thread or process that requests the *IO* information will not be blocked. It sends the request for the *IO* to the corresponding *IO* handler, then hangs up the *IO* requested task and start to handle other tasks that are waiting for processing. When the *IO* handler finishes the input or output operations and responds to the requesting process or thread, callback functions will be triggered, and the task requesting the *IO* will become ready and put into the waiting list to be processed. In this way, the system will be kept busy and all tasks will be served if the computer processors are powerful enough. In web applications, the server handles the requests from clients and requests to the database using the Non-blocking *IO*. When a task that requires input from the client side, the request for input will be send to the client side and a callback function will be attached to that request upon finishing. Then the server thread starts to handle other tasks. When the input from the clients is finished, the call back function will be called and the task will be marked as ready and put into the waiting list of the thread. When there is a request for querying the database, instead of blocking into waiting the response from the database, the sender thread will only send request to the database handler, then starts to process other tasks and returns to handling the database requested task when the response from the database is ready. As what can be seen from the two handling cases, the processor is kept busy and the system will generate the largest throughput. This mechanism suits for groupware tasks specifically, since there will be so many requests for the coordination information and the requests for database queries will also be abundant.

3.5 System Scalability

Web applications are different from traditional software applications, in which most of the calculation or processor related work are done in one single processor and there is only request for enhancing the processing power of the user's computer. Web applications put most of its heavy tasks on the server side, and the server is responsible for all service requests from various clients. Hence, the requirements on the processor of server machine are much higher. Also, even if web applications get some powerful enough processors right at the beginning, there will still be possibilities that they will run out of resources, as the number of clients using the service is increasing. So enhancing the power of processors is not a good solution in web applications. What web applications require is the ability to scale the system in the future [39]. As described above, groupware applications will require even more resources from the server, so the scalability of the application is even more important.

The scale of the system basically means adding more servers to the system and forming a cluster of servers. The cluster of servers can work in a distributed way to handle various services requested by the clients. Based on the processing time of each service, some services can be handled by the same server or the same service may run on several servers. Also, when appropriate load balancing algorithms are used, the number of servers running at various time periods can diverse based on the service requests to the server cluster. The scalability of the server makes the system extendible based on the work load. For groupware components, services can even be run in a separated server or several servers, to make it independent of the business logic parts.

According to [20, 39], to enhance the scalability of the system, the developers need to address several challenges. Firstly, it is necessary to defining the *API* between each components or servers. Since components are scaled into different servers, the call of methods or service will be affected by the calling of the *APIs* of the server. As a result, to enhance the scalability of the system, the calling of the *API* needs to be based on the calling of URIs, even in the same application. Otherwise, the deployment of different components into different severs will crash the whole system. Secondly, the load balancing algorithm should be carefully designed. The main purpose of scaling the system is to solve the performance bottleneck. However, if not distributed correctly, some service requests may be directed to the same server and cause even severe performance problem. To correctly distribute the application, developers need a deep insight into the architecture of the application, and have

to separate those intensive tasks into different server components. Thirdly, it is the consistency of data information between components. It is quite easy to ensure data consistency in a single server application, since the data is stored in one single machine and shared by each component, but, when system is scaled, the data may become inconsistent between components. The way in which data is shared between components is much more complicated in a distributed system than in a single server application. To ensure the data consistency, a mechanism targeted at controlling update of data between each component should be used.

3.6 Real-Time Cooperation

Group tasks require active involvement by each user. Each joined user needs to know what has been done by others and what remains to be done, so they can focus on the task and work together to make the group task progress efficiently. This basically requires cooperation to be real-time, which means that the client side needs to react quickly whenever there is an update on the server side [15, 20]. The client side can not work in a way that keeps refreshing the page to get the latest updates of the group task, as it is inefficient and distracts the user's attention on the task. The communication between the clients and the server needs to work in two directions. The clients can send inputs to the server and the server can also push the updates from other users to the clients. Traditional HTTP protocol will not work in this case, since it only supports client requests to the server or server sending response back to the client, while no support for server pushing information to the client side. Also, HTTP protocols have network speed limitations [20], which will limit the messages sent between the clients and the server. Though in normal cases this will not be a problem, it will affect how many users can take part in a certain group task in the future. Traditionally, there is no support for server pushing information to the client side. When the client side needs the information from the server side, it must send a request first to the server and after the server receives the request, the client will wait for the processing of the data. Only when the data is ready, the data can be sent to the client side. Usually there is also page refreshing on the client side related to the response from the server side. Definitely this approach will not work for groupware application, whose client pages should not be refreshed while the users are focusing on their work. The real-time requirement of the groupware application pushes the needs for the asynchronous client front end [10, 17]. Firstly, the client should not be blocked if waiting for certain response from the server side, or the client should not wait for the update

responses from the server side. This is quite clear, since the arrival time of updates from the server side can not be expected. If the client side works in a synchronous way, the client side will always be in a blocking situation. Existing techniques for the client side are the *AJAX* technique and *Web Sockets* technique. Both of these two technologies free the client side from being blocked by waiting for the response from the server side. Secondly, the server should push the updates to the client side, whenever it gets an update from a client. The challenge here lies in the detection of updates. Using polling techniques, where the server continuously keep checking for updates, will waste the resources of the server and make the server inefficient. The server should work in a reactive way, which means whenever there is an update from a client, the server will react to this change and act according to the corresponding business logic. So, for the server side, and reactive architecture is required.

3.7 High Availability

Our group application will finally be used by companies and they will use the system for business related purposes. The application is basically used in two situations, thus helping the company employees work together and supporting employees from different companies working together. The crash of the system would have severe negative effect on both of these two aspects. The company employees could not work together and the cooperation between different companies can not continue. So, the failure of the system will definitely cause the loss of profits for all companies that are using the application. Hence, it is strictly required that the system should work no matter what happened, i.e., the high availability of the system is essential.

The high availability of the system requires the server side to be always on. Hence the code of the system needs to be bug free and handle all exceptions appropriately. However, even if the system is totally bug free and works under all situations, the system might still crash due to a problem of the hardware that the system is running on [39]. Hardware fails with certain probability, no matter how well the hardware is designed and manufactured. In normal life, this will not be a problem, since the devices we are using work much shorter time period compared to servers and the restart of the devices will not cause too much effect on our daily use. But for servers, which generally work all day long after started, the confront of a failure of the hardware is almost inevitable. To improve the availability of the system, a distributed architecture is required. Using the distributed system, one application can be run on more than two devices. When one of the devices fails, the request

to that device can be redirected to another device that can offer the same services.

The idea of using distributed systems to improve availability is quite simple, but to implement a distributed system that works in this way is very challenging [6]. Firstly, there is a need for a load balancing algorithm that decides to which device a request should be redirected. Load balancing algorithm not only serves the purpose of balancing the work load between devices, but it also decides the route in case of failing devices. The algorithm needs to redirect the request and still make the workload between devices balanced. Secondly, the failure of the device needs to be monitored. When a device fails, the root of system needs to be informed, otherwise the requests will still be sent to the failed device. Also, the status of the failed device needs to be stored and transformed, so that the information of the users can be extracted after the requests being redirected.

Chapter 4

System Architecture

For groupware applications, the architecture means the way each part of the software is organised. In analogy to the architecture design in software, groupware can also be divided into different parts based on the key features of each part. According to [32], to build solid software, architecture is an essential part and has a profound effect on how many implementation challenges developers may meet during the development process. Also, the architecture of a software affects the performance of the software and generally can decide where the bottleneck might be in the whole system. Finally, when proposing an architecture for certain kind of application, designers should also take the scalability of the system into consideration, so when the work load become intensive in the future, the application can still work well or be extended easily.

According to [36], architecture is the most important part of software, as it works like the skeleton of the application. When there are some problems or failures of the application's architecture, in a sense, it might fail the whole application, no matter how perfectly the developers implement each part of the architecture. However, architectures are also the most difficult part to design for developers. In software companies, software architects are software developers who have abundant experience in software development and implementation. They are people who have been working in software development field for several years, who have already taken part in or lead the development of several projects, and who also have a good knowledge of the business logic in specific fields. As one can see, the requirements on software architects are really high. The seriousness of companies in hiring software architects also supports the view that architecture design in software development is really an important issue.

For groupware development, the architecture is even more important. Unlike other applications, the architecture of groupware has more effects on the

difficulties of the implementation part. According to [7, 18, 32], the reason is that the architecture of groupware will affect the placement of the concurrency control part in the system. As groupware applications generally have client side and server side, which side should be responsible for the implementation of concurrency control part pose have totally different challenges to the implementation of the concurrency control. When the concurrency control component is put on the server side, since the server itself is kind of synchronised critical section, the implementation of the concurrency control component is quite easy. However, when the concurrency part is implemented on the client side, the implementation becomes more challenging, since no global time sequence for asynchoronous operations exists at all. Also, the choice of the architecture will affect the performance of the system more, since the groupware applications have coordination information required to be transformed between clients.

In this chapter of the thesis, we will firstly discuss components that the application should contain and the main tasks for each part. Then, we have a very detailed design and analysis of the central architecture and the replicated architecture that are adopted to our application. Finally a architecture of the central and replicated architectures is proposed for our application, including the analysis of this architecture from both the performance and implementation perspective.

4.1 Components

Our groupware application is a task queue shared between different clients. Each client should have the same view of contents in the task queue after login. Each client can do the approved operations to modify the task queue. Clients that take part in the group task after the start of the group task, should have the same view of the task queue content with all clients that have been working in the group task. Based on all these requirements, the whole system can be divided into the following parts:

- *View update component*: This component is responsible for updating the view for a client side. This component will take input of the operations from the user, modify the contents of the client's task queue based on the verified operations, and display the updated queue content on the client's screen.

- *Lock apply component*: This component is located in both client side and server side for centralised architecture. It is responsible for acquiring the lock for modifying the content in the shared task queue for

client side. When the request for a lock is successful, the client will get the authority to modify the content in the task queue.

- *Login component*: This component is used for the login of clients. When a client logs on the system, this component will send the login events to the server, get the initial information back from the server, and initial the view content for the client.

- *Login manager component*: This component is responsible for the management of the clients that are currently online. It will be responsible for solving any late joining issues to ensure a consistent view for all clients. A list for online clients is kept by this component to keep track of all clients currently online.

- *Lock management component*: This component is used to maintain the mutual accessing of lock. It is this component that implements the concurrency control for the groupware application in the centralised architecture. This component will block those lock requests from clients if the lock is not available and hence a global sequence of the operations is generated.

- *Notification management component*: This component is responsible for the notification of each client. Whenever there is a modification in the shared task queue, each client that is contained in the list of clients currently online client list will get notified. The notification management component will also resolve conflicts with the login control component, to ensure consistent views in all clients after login.

- *Operation transform component*: This component is responsible for the transformation of operations coming from different clients in the replicated and combined architecture. This component is unique for replicated and combined architecture, which have no lock for concurrency control. The operation transformation component will implement the concurrency control in clients based on the operational transformation algorithm.

Not every architecture we adopt will implement all the above components. Also in different architectures, the ways how these components are going to work are different.

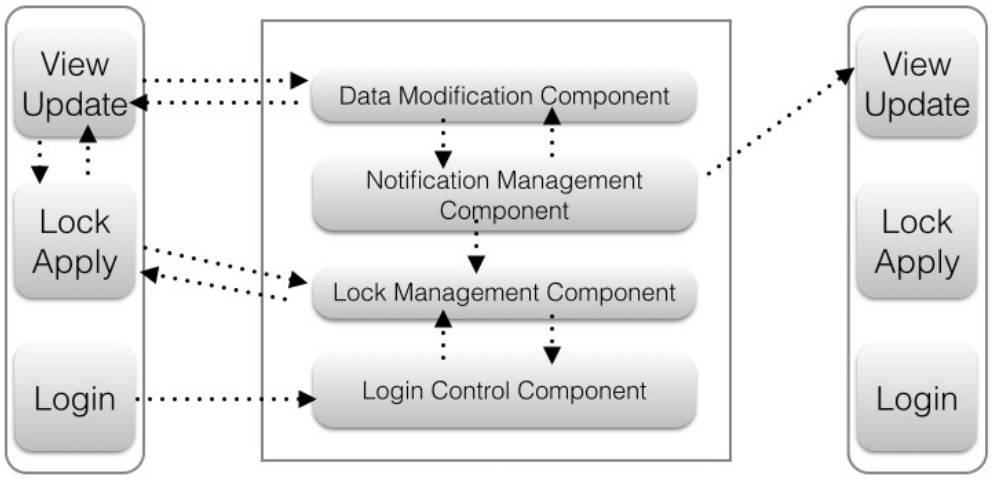

Figure 4.1: Centralized Architecture

4.2 Centralised Architecture

A centralised architecture has a central machine running a single application that takes care of input and output. In centralised architecture, the server will treat the input from the client side as synchronised operation sequence and it handles operations one after another. At the client side, the application only needs to receive the input from the client user, send it to the server, and when there is a response from the server side, display it. The centralised architecture for the shared task queue will look like the architecture showed in Figure 4.1.

The flow of information in the whole architecture can be divided into two parts, the *login sequence* and the *view updating sequence*. In the centralised architecture, when a user logs on the system, the *login component* in the client side will be triggered immediately. Then the *login component* will get the user's private information and send a request to the server. On the server side, this event will trigger the *login management component* to work. What the *login management component* will do on the server side consists of two steps. Firstly, it requests the lock from the *lock management component*. If the lock is free and the *lock management component* successfully returns the lock, it will continue to do the second step. If the lock is not free at the moment, the *lock management component* will give it the highest priority for getting the lock to ensure that it will get the lock immediately when the lock is free. Secondly, the *login management component* will access the newest

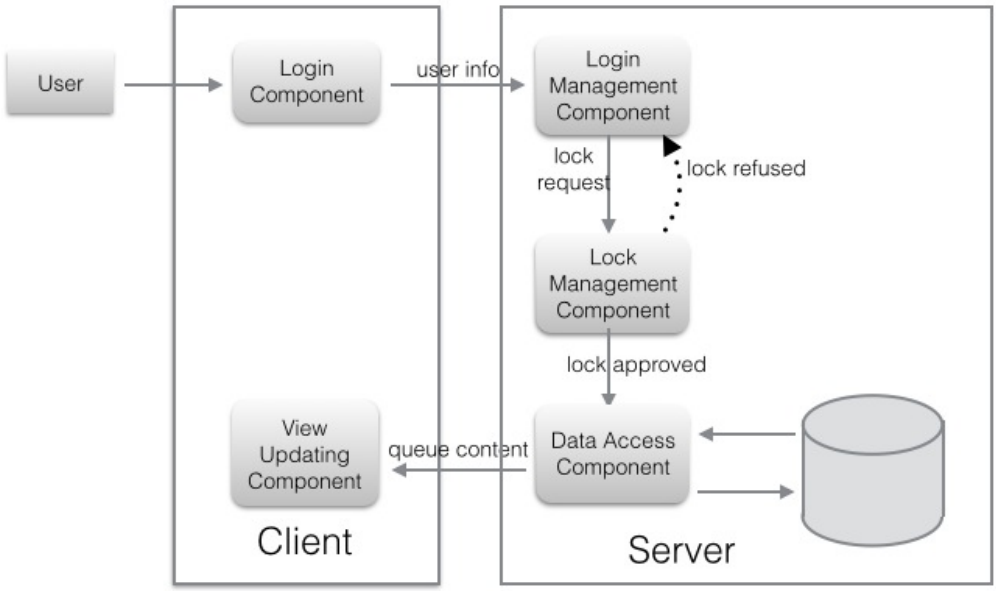

Figure 4.2: Login Sequence in Centralized Architecture

version of the shared task queue content, send it back to the client side as a response and then free the lock. After the *login component* on the client side receives the response from the server, it will initialise the client view based on the response and the thus *login sequence* has been finished. The whole process is described in Figure 4.2.

For the *view update sequence*, it starts when a user tries to modify the content of the shared task queue. The intention of modifying the shared task queue content will trigger the *view update component* on the client side immediately. Then the *view update component* will firstly send a request to the server to require the lock for modification. The request sent to the server side will be handled by the *lock management component*. A lock is approved by this component if it is free, otherwise the server side will deny the modification. The result of requesting a lock will be returned to the client side immediately, no matter the lock is approved or refused. Based on the result of acquiring of the lock, there are two branches for the *view update component* to continue. The first case, when the *view update component* receives the approval of the lock, the user can start to modify the content of the shared task queue. When the modification finishes, the *view update component* will send the modified content to the server, which will then update the shared task queue content, trigger the *notification component* on

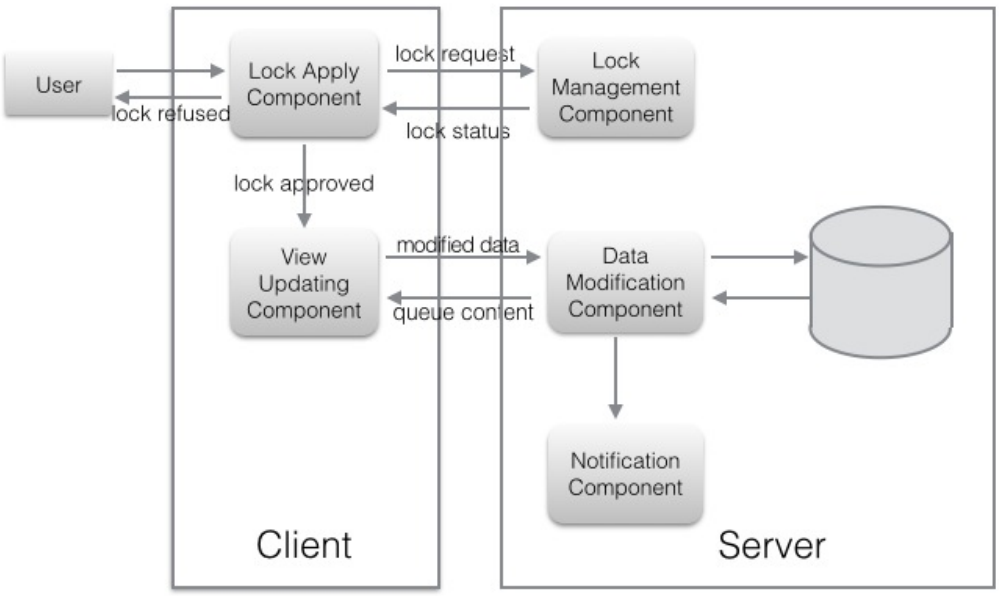

Figure 4.3: Updating Sequence in Centralized Architecture

the server to send the shared task queue content to all clients online, and free the lock. The *view update component* in each client will then update the view after getting the notification and the *view updating sequence* is completed in this case. The second case, when the *view update component* receives a refused result of getting the lock, it will block the user from trying to modify the content of the shared task queue until a notification of a new modification of the shared queue comes from the server side. Once the *view update component* has refreshed the view of the shared task queue based on newly received content, the client is freed from blocking and it can continue to request a modification. Hence the *view updating sequence* is done. The time sequence diagram of the *view update sequence* is described in Figure 4.3.

The advantage of using centralised architecture is obvious. It is absolutely easy to implement. The concurrency control takes full use of the single machine feature of the server and generates a global sequence for the distributed modifications that came from various client side. When the global sequence of operations exists, conflicts about operations can be avoided. The clients in different machines work as if they are operating the server machine in different time periods. The implementation of lock management in the server, which generates the global sequence, is quite easy, and developers working with concurrency control in database systems have accumulated abundant

experience in such aspects. For the client side, the acquisition of the lock has no difference compared to sending normal requests from the client to server. So, the shared task queue application based on this kind of an architecture can greatly reduce the workload for groupware developers.

However, the disadvantage of using centralised architecture is the performance. Firstly, the server has too much burden. Whenever there is an update from any client, the server needs to send the notification of the updated content to all other clients. When the number of clients increases, this definitely will be a bottleneck of the performance. Secondly, it is the network bandwidth requirement in the server side. As described above, each time there is modification, the server needs to send a new version of the content to all other clients. This is going to make the throughput of the server small, especially when there are lots of tasks in the shared task queue. Thirdly and most importantly, it is the blocking of the user that will cause serious user experience problems. When a user tries to log on the system, he might get blocked by another user who is modifying the content of the queue. Only after the modification is finished, the user can get logged in. Also, all other clients will be blocked by a client who is modifying the queue content, making the groupware application not different from a single machine application.

4.3 Replicated Architecture

Different from the centralised architecture, replicated architecture does not have a central machine. As the name of this architecture suggests, replicated architecture will have a copy of the same program on each client side. The operation of each client will be notified directly to other clients. The coordination of clients' behaviours will also be resolved on the client side. Most importantly, the client side is also responsible for handling the conflicts of the operations from other clients. The replicated architecture for our shared task queue application is shown in Figure 4.4.

The flow of the information in the whole architecture can also be divided into *login sequence* and *view updating sequence*. Like in the centralised architecture, the *login sequence* starts when a user logs on the system. The *login component* in the client side then starts to work by sending a login request to the server, wrapped with the client's private information and network address. On the server side, the *login management component* will be triggered by the request immediately. The server then sends the event of joining the new user to all other clients online, including the network address of this client and then, the private information of the client and the network related

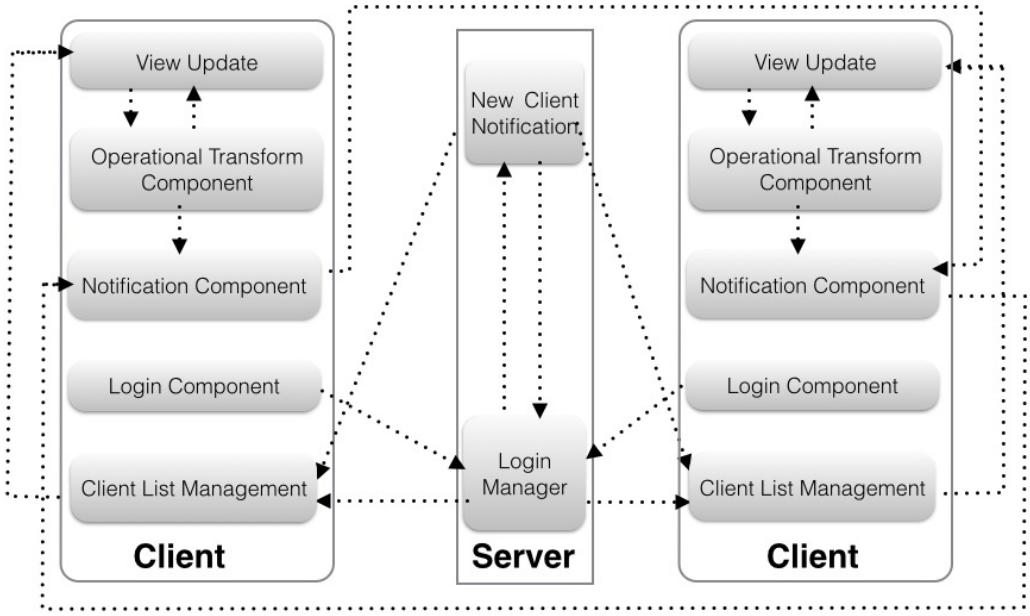

Figure 4.4: Replicated Architecture

address will be kept by the server. For any other clients that are online at the moment, when received this notification of newly joined user, the *client list management component* on the client side will store the information of the newly joined user on the client side. Then, the *client list management* will trigger the *notification component* to send a view initialisation event to the newly joined user, including the network address of the client. The initialisation events from various clients will be received by the newly joined user and the *operation transform component* of the new user's client side will transform all those initialisation events into a proper sequence of operations and then trigger the *view update component* to initialise the view. Then the network address of other clients will be kept by the new joined client's *client list management component* and the *login sequence* is performed for the new user and the other clients will send the modifications to this client when some operations are done and the clients can start to send modifications to other clients too. The sequence of the operations is illustrated in Figure 4.5.

The *view updating sequence* starts when a user tries to modify the content in the shared task queue. The modifying operation will be directly sent to the *operation transform component*. Based on the algorithm in *operation transformation component*, the operation will be transformed into another appropriate operation. Then the transformed operation will be handed to

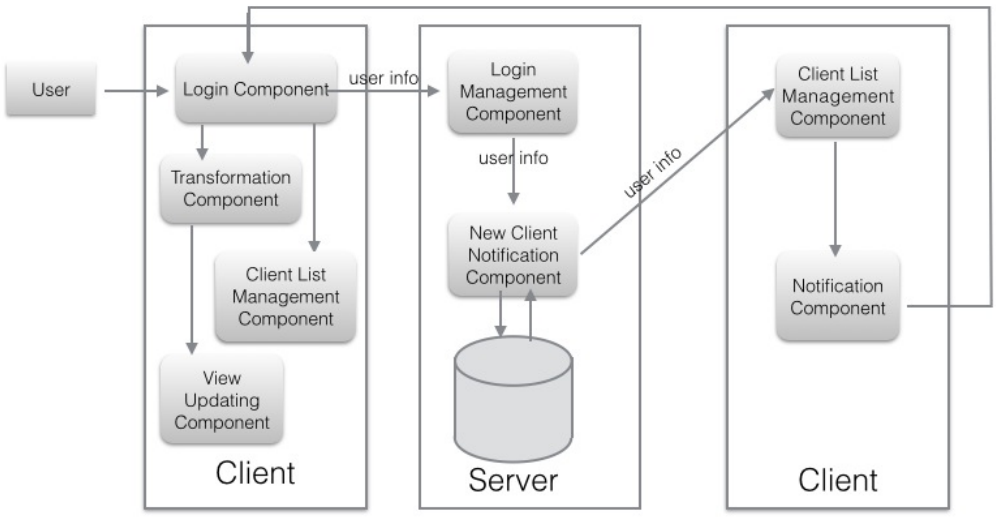

Figure 4.5: Login Sequence in Replicated Architecture

the *view update component* again and the *view update component* will update the view based on the transformed operation. Also the transformed operation will be sent to other clients that are kept in the *client list management component* by the *notification component*. When other clients received the operation, the received operation will also be sent to the *operation transform component* for doing some transformation again. Only after the transformation has been done, will the new transformed operation be transferred to the *view update component* and the view of that client will be updated. Then the whole sequence of the view updating has been carried out. As we can find from the whole process, there is no lock like concurrency control part in the architecture. This is because the *operation transform component* will handle the concurrency control based on its algorithms, generate a global sequence of the operations, and solve conflicts between clients. The whole process is shown in Figure 4.6.

The advantages of using replicated architecture are manifold. Firstly, the server is freed from sending notifications to each clients every time there is a modification from some client. As a result the server can focus more on the business logic of the application. Secondly, data transfer is reduced. In the replicated architecture, the data sent in the notification concerns only the modifying operation rather than the whole content of the shared task queue. Hence the transfer load is not related to the number of tasks in the shared queue. Finally, and most importantly, no users is blocked by the system from

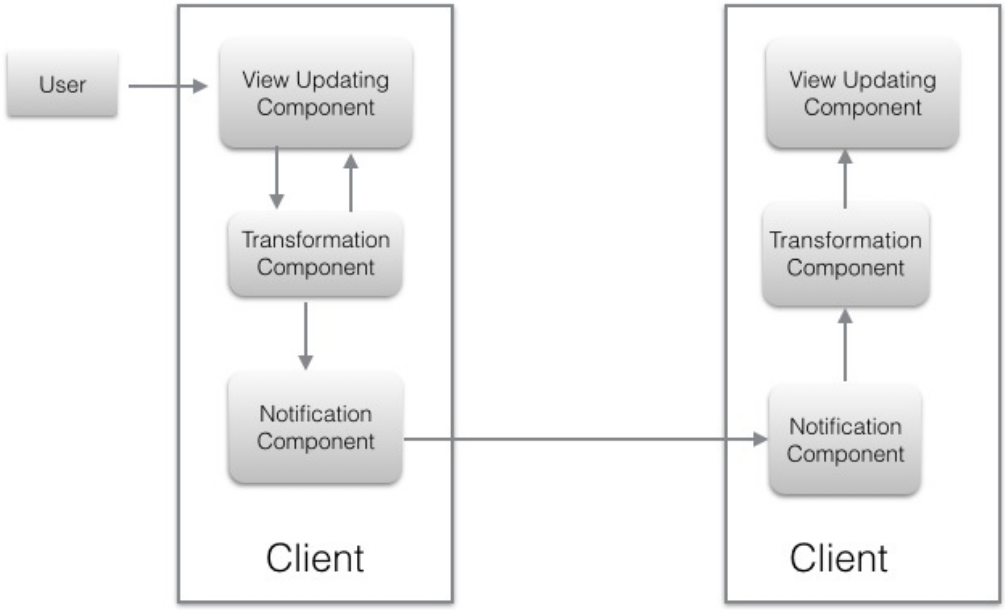

Figure 4.6: View Updating Sequence in Replicated Architecture

updating the content in the queue. The modification of the content in the task queue can be seen almost immediately to the client users, which is a very impressive user experience improvement. So, if the replicated architecture is used, the user can focus more on the task itself rather than being blocked by the system.

The disadvantages of the replicated architecture are also obvious. It is far more complicated to implement than the centralised architecture. For the login sequence, the initialisation of the view for the newly joined client is much more complicated than that in the centralised architecture, since the client needs to resolve conflicts under the situation where no global sequence is offered. For the view updating sequence, the stability of the program greatly relies on the implementation of the operation transformation component, which is a very complicated algorithm and not easy to implement at all. Also, the use of the replicated architecture will introduce data inconsistency for some short periods of time. The reason for this is that the operations of the shared queue are transferred through the network. The arrival sequences of operations vary between from client to client.

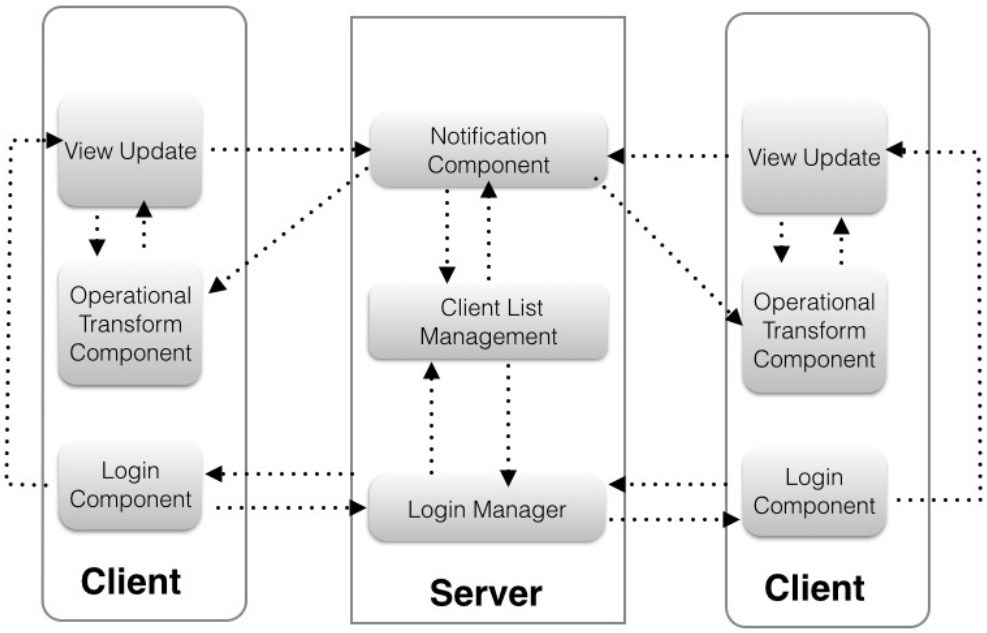

Figure 4.7: Combined Architecture

4.4 Combined Architecture

As we can see from the description above, the strenghts of both centralised and replicated architecture are opposite to each other. Replicated architecture has the advantage in aspects where centralised architecture have drawbacks, vice visa. So, it is natural for us to think about combining these two architectures together and forming an easy to implement yet well distributed architecture. The main disadvantage for replicated architecture is the implementation of the concurrency control algorithm to generate the global sequence, yet the centralised architecture can generate the global sequence without much effort. For the centralised architecture, its main disadvantages are the blocking of the client and the network transformation bandwidth, while using the replicated architecture's *operation transformation component*, the user will not be blocked and the bandwidth cost is low. So, the combined architecture will take use of the centralised architecture's global sequence of operations while using the *operational transformation algorithm* to unblock the client. The architecture is shown in Figure 4.7.

The two information flow sequences, i.e., the login sequence and the view updating sequence, change a little bit in this architecture too. The login

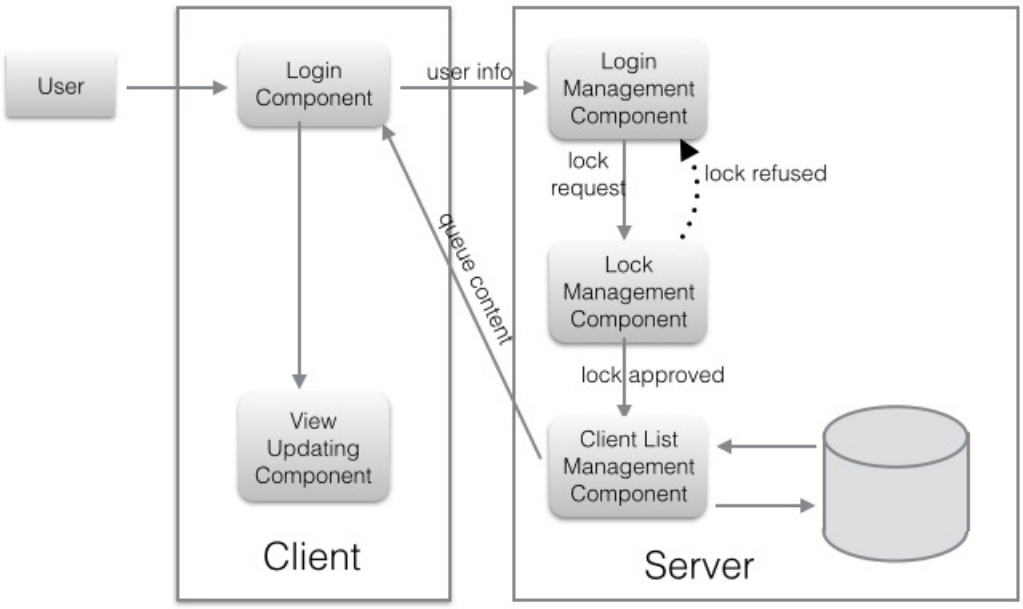

Figure 4.8: Login Sequence of Combined Architecture

component is still on the client side and it will send the login request to the server when a new client tries to login. The login management component, located on the server side then will trigger the client list manage component to add this new client. The newest version of the shared queue content is abstracted from the server, the process of which will be mutual with the process of data modification component to guarantee that the newest version of the data is acquired. Then the login manage component will send the shared queue content back to the client and the login component will use the data to initialise the client view. The whole process is much simpler than both the centralised architecture and the replicated architecture. The process is showed in Figure 4.8

For the view updating sequence, the client can immediately change the view of the client whenever users try to modify the content in the shared task queue. The modification operation of the content will be sent to the server. On the server side, the operation transform component will handle this request and transform the operation based on the transform algorithm. The transformation algorithm here can be quite simple, since the global sequence is guaranteed by the arrival time of the operations to the server. The only task for the transformation component is to change the operation to the newly arrived operations. If the operation is modified, an undo response

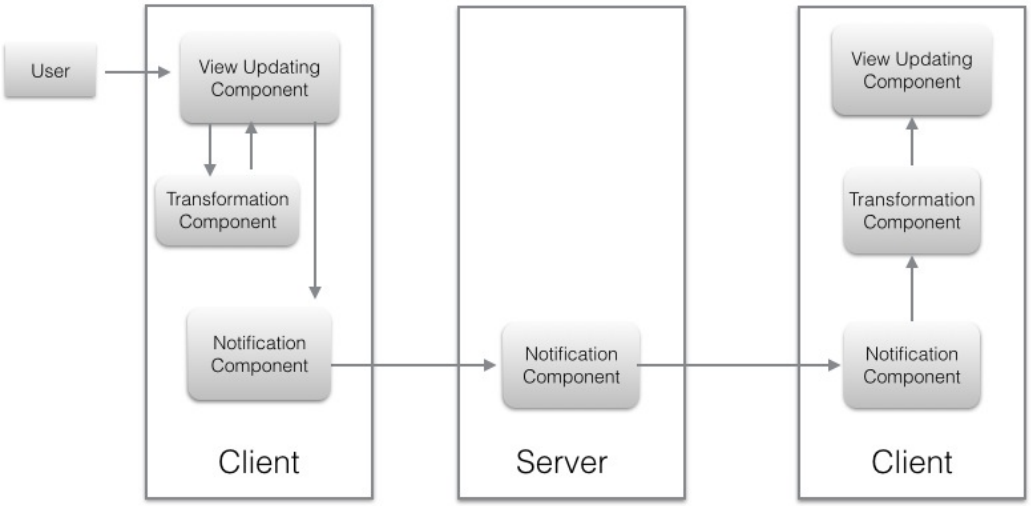

Figure 4.9: Update Sequence of Combined Architecture

will be sent back to the client side, which will then undo the modification of the previous operation. Then the server will trigger the data modification component and modify the data in the shared task queue saved on the server side. Finally, the notification component will be triggered and the transformed operation will be sent to each client. Then the clients can update their view of the shared task queue by performing this operation and the sequence of view updating is done. The whole process is described in Figure 4.9

As we can see from above, the data transformed between the client side and the server side are only operations for modifying the shared task queue content, rather than the whole content of the task queue. Hence the bandwidth cost for each request and response is relatively smaller and will not grow along increasing content in the shared task queue. The client will not be blocked when trying to modify the content in the shared task queue, as the modification can be performed immediately after the user initiates the operation. It is the system that will finally re-modify the content of the queue to ensure the consistency of the contents. This will work similarly to case in the replicated architecture, giving a very impressive user experience improvement. Also, for the complexity of the implementation, we can get the answer through describing the information flow sequence. Both of the two operation sequences are simplified and since the transformation algorithm is a simplified version, the implementation of the operation transformation will be much easier than the implementation of the transformation algorithm in

the replicated architecture.

The drawback of this architecture is, as can be found from the figure, the burden on the server side. Compared to the centralised architecture, which has drawbacks in the server workload, the server in the combined architecture has a transformation component added. So, theoretically the burden of the server in the combined architecture might be heavier than in the centralised architecture. Also, the inconsistency of the task queue content is not solved in the combined architecture since the data transformed through the network consists of modifying operations rather than the task queue content itself.

Chapter 5

Concurrency Control Algorithm

Concurrency control is a necessary part in the development of a groupware application [15]. Its purpose is to solve any conflicts caused by distributed users and to enable participants tightly cooperate with each other. The ways in which developers solve conflicts or control concurrency vary. Techniques such as explicit locking or transaction processing have been used in the development of database applications to solve concurrency control for decades. However, the traditional ways of concurrency control seem not to work well in the case of groupware applications [15].

There are many issues that need to be taken into consideration when designing a concurrency control algorithm for a groupware application. Firstly, the target of offering a consistent view for each client should be considered, i.e., the WYSIWIS (What You See Is What I See), which is necessary to ensure the progress of the group task in the correct direction. The cooperation between a group of users will end up in a chaos if some of them sees a slightly different or out-of-date version of the data. Secondly, the response time from the server is also an important consideration in groupware application. The response time basically includes the time it takes for the client to access data, update data, and broadcast those changes to all other clients. The concurrency control method in the database situation like transaction will not work well here, because blocking some user from providing information to synchronous concurrent operations originating from different clients will increase the response time [15].

There are several approaches that have been used to solve the concurrency control problem, but most of them are not well suited for groupware applications. In [15], several traditional methods that are used for concurrency control have been discussed. Locking, for example, is a concurrency control solution that locks the data before it is updated. This method is easy to understand and not difficult to implement for concurrency control. The

drawback of this approach is that it requires an overhead time to send the request of a lock, which will add extra time to the response procedure. Also, the position or the object that should be locked is hard to define. There are cases in which deciding what should be locked is difficult. Finally, the time that when the lock should be released or requested is also not clear. The moment for requiring and releasing a lock will affect the efficiency of the operation execution and the correctness of the execution result.

Another approach for concurrency control is transaction, which has been used in interactive multi-user systems. However, in the case of groupware application, using transaction for concurrency control will inevitably increase the complexity in algorithm implementation and increase the time for executing operations. Also, the way we choose to implement transaction mechanism such as locks, will bring other problems related to the technique.

In this chapter, we will introduce a concurrency control algorithm for groupware application that is adopted from the *Operational Transaction Model (OT)*. This chapter will be organised as following: Firstly, we introduce the *Operational Transaction Model*, including definitions and a *Generic Operational Transaction* (GOT) algorithm. Then the concurrency control algorithm adopted from *Operational Transaction Model* will be presented. Finally, we give a detailed explanation of the adopted algorithm.

5.1 Operational Transformation Model

5.1.1 Challenges in Concurrency Control of Groupware

Operational Transformation Model addresses the concurrency control problem in groupware and its target is to keep the content in each client consistent. Before we explain how *OT Model* works, it is better to discuss what are the problems groupware developers meet in consistency maintenance aspects. According to [38], there are three challenges in consistency maintenance. To illustrate these three problems clearly here, Figure 5.1 is used.

Divergence: Since users of a groupware are distributed over several machines in most cases, operations from the same client may arrive at other clients with different delay time periods. Different arrival delays of certain operations among different clients usually result in various sequences of operations at each site. Hence, the final execution result of various operation sequences will not be identical, unless all the operations are commutative, which is generally not the case. Let us take the *classic operational transformation puzzle* from Figure 5.1 [16] as an example. The observed operation

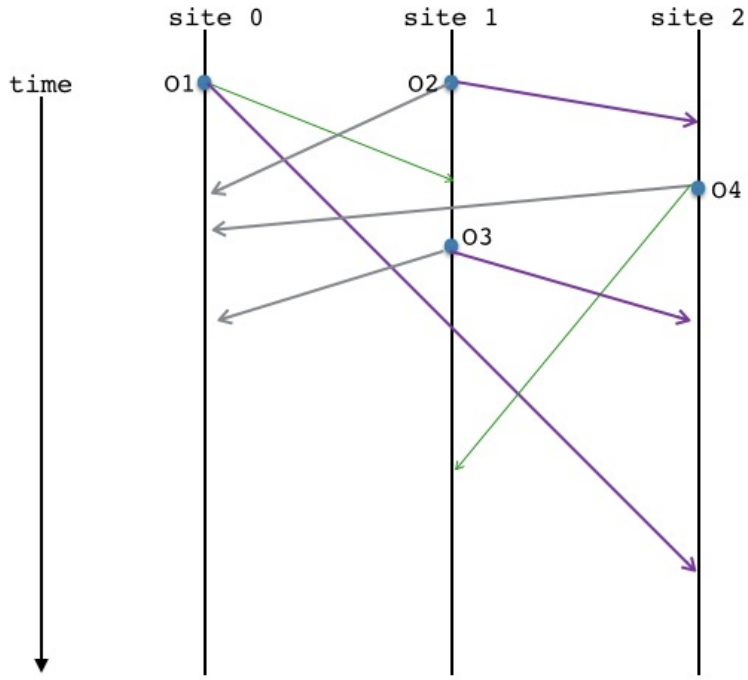

Figure 5.1: Classic Operational Transformation Puzzle

sequence at site 0 is O_1, O_2, O_4, and O_3; at site 1, the sequence is O_2, O_1, O_3, and O_4, and at site 2, the sequence is O_2, O_4, O_3, and O_1. Since most groupware require the consistency of final result, the divergent final result is not acceptable.

Causality Preservation: As a result of the nondeterministic communication latency, operations from the same site may arrive at certain sites in a sequence that is different from the cause-effect (some operations may depend on the previous executed operations) order. Think about the operations in Figure 5.1: Operation O_3 is conducted by certain user, when the arrival of O_1 has been seen at site 1, hence the execution order of O_3 should be after O_1 in all sites. In a sense, O_3 is dependent on O_1. However, as a result of network communication latency, the arrival of O_3 is before O_1 at site 2. This is possible since O_1 is transferred between site 0 and site 2 while O_3 is transferred between site 1 and site 2. The network route is different. If O_3 is an operation that targeted at modifying the effect caused by O_1, then the arrival of O_3 at site 2 may cause confusion to the user, hence a *Causality Violation* exists.

Intention Preservation: Since groupware is a multi-user system, the

generations of operations are highly concurrent. As a result, the context where some operation was generated may be changed when the operation is executed. Therefore, the intended effect of the operation may be different from the actual effect when the operation gets really executed. To illustrate this case, let us also take a look at the execution of operation in Figure 5.1. Operation O_1 at site 0 and Operation O_2 at site 1 are generated independently, without knowing the existence of each other. Site 0 will first execute O_1, before the arrival of O_2. When O_2 arrived at site 0, the context of the execution has been changed by the execution of O_1. Hence, the execution of O_2 may lead to a different result with its intension. Let us take a more detailed case into consideration. Before the generation of O_1 and O_2, the groupware is in a consistent state where both site 0 and site 1 are in the initial state "ABCD". Then O_1 = Insert["a", 1] is generated at site 0, which intends to insert element "a" in the second position, i.e., put "a" between "A" and "B". Also, O_2 = Del[1] is generated at site 1, which intends to delete element B at the second place. After the execution of these two operations, the expected content should "AaCD. However, the actual content at site 0, after the execution of O_1, is "AaBCD" and with the later arrival of O_2, the content is going to be "ABCD". The execution of operation O_1 changed the context in site 0, hence it is different from where O_2 is generated. As a result, the execution of O_2 violated its intended effect.

The challenge of divergence is supposed to be solved when there is a serialisation protocol [26]. The serialisation protocol can be generated by distributed algorithm or a centralised coordinator [15]. The use of serialisation protocol is to ensure that final results of different operation sequences are the same as if they were conducted under the same global operation sequence. The causality violation challenge can be solved, when we define the execution context of each operation well and delay the execution of some operations whose execution context is not ready. To address the problem of *intention violation*, the context under which the operation was generated should be preserved [38] and the transformation of operations should only be performed when the same execution context has been established.

5.1.2 Operation Transformation Algorithm

The algorithm in [38] is based on three methods to solve the three challenges described above. To solve the divergence problem, the state vector structure is used to indicate the total sequence of the operations and REDO and UNDO operations are defined to keep the distributed clients in convergence, whose definition will be given later. Causal precedence is preserved when the state vector is applied to indicate the current state of the client. The inten-

sion violation problem is addressed by the use of the GOT transformation algorithm.

A *state vector* was first proposed in [15]. It is an n dimensional array $SV[n]$, where n denotes the number of sites in the whole system. The value of $SV[i]$ is the number of operations generated by the site i and executed by the current client site. The *total sequence* of an operation in the global sequence is defined by $\sum_{i=0}^{n} SV[i]$, where $SV[i]$ is the value of $SV[i]$ when the operation was generated. If $\sum_{k=0}^{n} SV_i[k]$ of O_i is smaller than $\sum_{k=0}^{n} SV_j[k]$ of O_j , it means that the *total sequence* of O_i precedes O_j. If some operations O_j in client j have *total sequence* larger than a new operation O_i, then O_j should be undone and the client should be restored to the context when O_j was not yet executed. We define this procedure as *UNDO*. After the execution of O_i, the undone operations can be executed again with the execution effects of O_i, which is defined procedure as *REDO*.

Every client keeps a state vector, which records how many operations from other clients have been executed by the current client. When a new operation O_i, with a state vector SV_i arrives at client j, SV_j will be checked. If there is some component k in SV_j that $SV_j[k]<SV_i[k]$, it means that before the generation of O_i, client i has executed some operations from client k that has not been executed by client j when O_i arrived at client j, which is defined as O_i not being *causally ready* [38]. Then the *causality violation* can be avoided by delaying the execution of O_i until it becomes causally ready.

The GOT algorithm is based on the transformation of operations. There are basically two kinds of transformations, the *inclusion transformation* and the *exclusion transformation*. The *inclusion transformation* function is defined as $\mathrm{IT}(O_i, O_j) = O_i'$, which can only be executed when the generation context of O_i is equal to the generation context of O_j. After the execution of inclusion transformation function, O_i is added with the execution effect of O_j. Hence O_j is executed right before the execution of O_i, which is defined as O_j being in the *context preceding* O_i [38]. The *Exclusion Transformation* function is defined as $\mathrm{ET}(O_i, O_j) = O_i'$, whose execution condition is that the execution context of O_j should be the *context preceding* the execution context of O_i. The execution will exclude the execution effects of O_j from O_i, and after the execution of the function, O_i' and O_j are in the same execution context. The two transformation functions will meet the requirement of *reversiblity* [38] to guarantee the correctness of the transformation, where *reversibility* means that the consecutive execution of the inclusion transformation and exclusion transformation will add no effects to the target operation. Each operation that have been appropriately transformed will be executed in the client and stored in *history buffer* HB, which is a one dimensional array of executed operations. Based on these two transformation functions,

the GOT algorithm will get the correct execution form of operation EO_{new} in the following way(O_{new} is *causally ready* here):

- **(1)** Find the first operation in HB that is not in the *context preceding* O_{new}. If no such operation EO_k is found, return $EO_{new} := O_{new}$
- **(2)** Else, find all operations in HB that are behind EO_k in the *total sequence* but *causally preceding* O_{new}, and store them in a buffer of operations EOL in the order of *total sequence*. If no such operations exist, return $EO_{new} := \text{LIT}(O_{new}, [HB[k], HB[k+1], ..., HB[n]])$, where n is the size of HB.
- **(3)** Else, For EO_{c_i} in EOL, get EO'_{c_i}:
  - $EO'_{c_1} := \text{LET}(EO_i, [HB[k], HB[k-1], ..., HB[c_1 - 1]])$
  - for $i > 1$,
    - TO $:= \text{LET}(EO_{c_i}, [HB[k], HB[k+1], ..., HB[c_i - 1]])$;
    - $EO'_{ci} := \text{LIT}(\text{TO}, [EO'_{c_1}, ..., EO'_{c_i - 1}])$
- **(4)** $O'_{new} := \text{LET}(O_{new}, EOL'^{-1})$
- **(5)** return $EO_{new} := \text{LIT}(O'_{new}, HB[k, n])$.

In the algorithm LET and LIT are recursive functions defined for applying repeatedly exclusion and inclusion transformation [38]. The first step of the algorithm is to look for *independent* operations, which are operations that are not in the *context preceding* the new operation. When there are *independent* operations, it means there are operations in the current client site that have not been executed in the original site when the received operations generated. Hence, O_{new} should be transformed to include the effect of executing EO_k and any later operations. In the case that all operations behind EO_k in the *total sequence* are *independent* of O_{new}, it must be the case that the context of EO_k in the current client site is equal to the context of generating O_{new} in the original site, since EO_{new} is causally ready and all operations behind EO_k are not *casually preceding* EO_{new}. Hence the execution effect of EO_k in the current site can be included directly using *inclusion transformation*. And the later operations' effects can also be included consecutively. In other cases where there are several operations that O_{new} is dependent on, the context of executing EO_k is not equal to the context when O_{new} was generated in the original site. Hence, we should first transform O_{new} into the same context with EO_k, by excluding those operations' effects that O_{new} is dependent on but behind EO_k in the *total sequence*. Here, the excluded execution effects of

operations should be in the same operation effects of those operations when executed in O_{new}'s original site, otherwise the preceding operation would not be *context preceding* O_{new}. This is done by repeatedly excluding the operations in HB that are context preceding the operations in EOL, and then including those operations' effects that are causally preceding this operation in the original site of O_{new}. This is the step (3) in the algorithm. When all the operations that O_{new} is dependent on are transformed into the same operation in the original site of O_{new}, in step (4), O_{new} is transformed into the same context with EO_k, by excluding all the execution effects of the dependent operations. Then the repeated inclusion transformation can be done in step (5).

The integrated algorithm with UNDO/REDO scheme and GOT algorithm is as follows [38]:

- **(1) Undo** operations in HB from right to left until an operation EO_m is found such that EO_m is in the *context preceding* O_{new}
- **(2) Transform** operation O_{new} into EO_{new}, using the GOT algorithm, and perform EO_{new} at the current site.
- **(3) Transform** operation EO_{m+1} in $HB[m+1, n]$, where n is the size of HB, into the new execution form EO'_{m+i} as follows:
  - EO'_{m+1} := IT(EO_{m+1}, EO_{new}).
  - For $2 \leq i \leq (n-m)$,
    - TO := LET(EO_{m+1}, $[HB[m+1], HB[m+2], ..., HB[m+i-1]]$)
    - EO'_{m+i} := LIT(TO, $[EO_{new}$, EO'_{m+1}, ..., $EO'_{m+i-1}]$)
- **Redo** EO'_{m+1}, EO'_{m+2}, ..., EO'_n, sequentially.

The algorithm first checks the operations executed in the current client site that are behind the new arrived operation in *total sequence* and undoes all those operations. Then it transforms the newly arrived operation into the correct form based on the GOT algorithm and the transformed operation will be executed at the current client site. After executing EO_{new}, the undone operations should be redone. The operations should add the execution effect of EO_{new}. This is done by excluding the effects of the preceding undone operations and including the execution effects of EO_{new}.

5.2 Centralised Operational Transformation

In Chapter 4, we have proposed the combined architecture to solve the concurrency control problem, while unblocking each client. The use of centralised architecture can simplify the Operational Transformation algorithm a little bit, but the three challenges discussed above should also be addressed. The actual adopted algorithm will start from the *inclusion transformation* and *exclusion transformation* functions. To illustrate it correctly, we introduce it based on the Transformation Matrix Idea of [15].

5.2.1 Transformation Matrix

For our shared task queue groupware application, there are three kinds of approved operations, Insert, Modify and Delete. Inserting is the creation of new tasks and putting them into the shared queue. Modifying is about changing the content of the existing tasks and deletion means the removal of the existing tasks from the queue. So, the transaction matrix T would be a 3×3 matrix. Figure 5.2 gives a detailed description of each component in the transformation matrix.

		1	2	3
		Insert	Delete	Modify
1	Insert	IT11	IT12	IT13
2	Delete	IT21	IT22	IT23
3	Modify	IT31	IT32	IT33

Figure 5.2: Transformation Matrix Index

Figure 5.2 shows how the operations are mapped to the corresponding numbers. So, in the following definition of transformation components, $T21$ means the transformation of receiving a Delete operation from the remote server when there is a Insert operation that has been executed in the local client, which has not been executed in the sending client when sending the Delete operation.

Listing 5.1 describes three transformation components which will transform the corresponding operations into the correct operation when the local

Listing 5.1: Transformation of Insertion Operations

```
IT11(Oi, Oj, Pi) = Oi' where
  if (Xi < Xj)
    Oi' = Insert[E, Xi]
  else if (Xi > Xj)
    Oi' = Insert[E, Xi + 1]
  else if (Pi == 1)
    Oi' = Insert[E, Xi]
  else
    Oi' = Insert[E, Xi + 1]
  fi

IT21(Oi, Oj, Pi) = Oi' where
  if (Xi < Xj)
    Oi' = Delete[E, Xi]
  else
    Oi' = Delete[E, Xi + 1]
  fi

IT31(Oi, Oj, Pi) = Oi' where
  if (Xi < Xj)
    Oi' = Modify[E, Xi]
  else if (Xi >= Xj)
    Oi' = Modify[E, Xi + 1]
  fi
```

client has executed an insertion operation that have not been executed when the sending operation was sent in the sending client. $T11$ is when the received operation is an insertion. Then, if the insert operation that has been executed in the receiving client is inserted before the position where the received insertion should be taken place, the received insertion should be inserted in the position after its original position. For other places of the executed insertion, the received insertion will not be affected. The Delete and the Modify will be transformed in the same way as the insertion.

The definition of the three components of transformation when the executed operation is the Delete operation is shown in Listing 5.2. The situation of the delete operation that has been executed before the receiption of a remote operation is quite similar to the case of the insert operation. Instead of moving the received operation to one place after the original, it moves the original operation to one place before the original place. However, when the received operation will be executed in the same place as the executed delete operation, the late coming operation will be ignored. This is because the deletion of a task will make all other operations for that task meaningless.

Listing 5.2: Transformation of Deletion Operations

```
IT12(Oi, Oj, Pi) = Oi' where
  if (Xi < Xj)
    Oi' = Insert[E, Xi]
  else if (Xi >= Xj)
    Oi' = Insert[E, Xi - 1]
  fi

IT22(Oi, Oj, Pi) = Oi' where
  if (Xi < Xj)
    Oi' = Delete[E, Xi]
  else if (Xi > Xj)
    Oi' = Delete[E, Xi - 1]
  else
    Oi' = EMPTY
  fi

IT32(Oi, Oj, Pi) = Oi' where
  if (Xi < Xj)
    Oi' = Modify[E, Xi]
  else if (Xi > Xj)
    Oi' = Modify[E, Xi - 1]
  else
    Oi' = EMPTY
  fi
```

The definition of the components when the executed operation is modifying is quite simple, as showed in Listing 5.3, because the modifying operation will not cause changes in the number of tasks. The only thing we need to explain here is the case when the two modifications modify the same task. In this case, both modifications will be kept for the task.

Above, we introduced the transformation algorithms for inclusion transformation. Exclusion transformation functions are quite similar to inclusion transformation, but with opposite effects. The code in Listing 5.4 implements the exclusion transformation function $ET11$.

5.2.2 Auxiliary Operations

LIT function is defined in Listing 5.5. When the list of operations whose execution effects need to be included by the received operation is empty, the received operation need not to be transformed at all. Otherwise, the operation will be recursively transformed with each operation's execution effect included.

Listing 5.3: Transformation of Modification Operations

```
IT13(Oi, Oj, Pi) = Oi' where
  Oi' = Insert[E, Xi]

IT23(Oi, Oj, Pi) = Oi' where
  Oi' = Delete[E, Xi]

IT33(Oi, Oj, Pi) = Oi' where
  Oi' = Conbine_Modify(E, Xi)
```

Listing 5.4: Exclusion Transformation of Insertion Operations

```
ET11(Oi, Oj, Pi) = Oi' where
  if (Xi < Xj)
    Oi' = Insert[E, Xi]
  else if (Xi > Xj)
    Oi' = Insert[E, Xi - 1]
  else if (Pi == 1)
    Oi' = Insert[E, Xi]
  else
    Oi' = Insert[E, Xi - 1]
  fi
```

LET function is defined in Listing 5.6, which works quite similarly to the LIT but with an opposite effect.

5.2.3 Adopted Operational Transformation Algorithm

The adopted centralised OT algorithm is shown in Listing 5.7. The initial part of the algorithm simply sets the auxiliary structures to the initial value. Here C denotes the current number of operations that have been received from the server and that have been executed. The initial value of C should be 0, meaning that no operations have been received and executed from the server yet.

The generation of operations simply gets the operation from user interface and wraps it with the current value of C. What should be announced here is that the generation of operations process should be mutually excluded with the process of execution of operations. If the execution of operations can run asynchronously whenever the user is inputing, then the input of the user might have a wrong base and the C value read from the client side might not be correct. The generated operation is pushed into the queue with the flag set to false, meaning that the operation is a local operation that has not

Listing 5.5: Definition of LIT Function

```
LIT(O, OL) = O' where
  if OL = []
    O' = O
  else
    O' = LIT(IT(O, OL[0]), Tail(OL))
  return O'
```

Listing 5.6: Definition of LET Function

```
LET(O, OL) = O' where
  if OL = []
    O' = O
  else
    O' = LET(ET(O, OL[0]), Tail(OL))
  return O'
```

yet been executed by other clients. Then the operation is sent to the central server which will notify all other clients.

Receiving operations from the server is simple too. All the received operations will be wrapped with the flag equal to true and pushed into the queue.

The most important part is the execution of the operations. As the centralised architecture guarantees the global sequence of operations, the algorithm can be simplified. Instead of using the *state vector* to denote the state of the client, the centralised algorithm uses the number of executed global operations to identify the state of the client. The causal violation is avoided, since every operation generated at a site will arrive other sites at the sequence of arriving at the central server. Hence, operations generated by the same site will arrive to other sites in the same order as in the generated order. So, each client site does not need to log how many operations from a specific site have been executed. There are three categories of operations the client may execute. Firstly, it is the execution of operations generated from local clients. The state of the current site is denoted by the number of global operations it has executed, i.e., C. Hence the algorithm first compares the number of global operations executed when the operation is generated with the current number of global operations that have been executed at the site. When there are operations with states that are after the generation of the local operation, the local operation needs to be transformed in case of a conflict with the executed operations. The transformation is based on the GOT algorithm, with small changes. The operations that are local, are operations

that have only been executed on this site and no responses from the server have been received yet. For a local operation, the operations that precede it would only be the local operations which also have not been confirmed by server yet. This is also correct, when we define it more generally: The operations that causally precede a newly arrived operation in the centralised architecture, would only be the operations that were generated at the same site as the arrived operation and were executed with global sequences behind the state when the arrived operation was generated. Therefore, before we send a local operation to the server, we have already known its causally preceding operations. To simplify the procedure of the transformation, we store all the local operations with its preceding local operation effects excluded into the local buffer LB. Hence, we can get EOL', a structure defined in GOT algorithm, by including all the global and local operations executed after the generation of a local operation. The required transformation of the new operation is then the same as with GOT algorithm. To reduce the transformation steps, we send the new operation with all local operation effects excluded to the server. No undo or redo operations are needed here, since newly generated local operations will be in the end of total sequence.

The second category is the operations received from the server, but with the client identification equals to the client. This means that the received operation is generated by the local client and has been executed or is going to be executed by all other clients. The operation should be changed into global operation and not local operation any more. Hence, the operation should be deleted from LB. Since no operation needs to be done here, no transformation, redo, or undo are needed.

The third category is the operations that are received from the server and were generated by other clients. The execution of this kind of operation is quite similar to the execution of local operation. However, since the operations received have already excluded all the effects of operations that are causally preceding it, the inclusion transformation function can be directly applied. No calculation of EOL' is needed. Before the execution of the transformed operation, *undo* operations are required. After performing the transformed operation, *redo* is executed. Being different from GOT algorithm in [38], redo is carried out by applying *inclusion transformation* function to the operations in current HB.

Listing 5.7: Integrated OT Algorithm

```
Algorithm:
    C <- 0
    Qi <- empty
    HB <- empty
    LB <- empty
  Generate Operations:
    receive operation o from user interface
    get C
    Qi <- Qi + <i, o, C, false>
  Receive Operations:
    receive <j, o, C, true> from the network
    Qi <- Qi + <j, o, cj, true>
  Execute Operations:
    For each <j, Oj, Cj, flag> belongs to Qi
    Qi <- Qi - <j, Oj, Cj, jflag>
    if (j == i && jflag == false)
      <Ik, Ok, Ck, true> <- entry with smallest k in HB,
          where ck > cj
  <Ot, Ct> <- first item in LB
  EOL' = empty
  do while <Ot, Ct> != null
    TO <- TO + LIT(Ot, HB[Ct, Ck - 1], 0, Ck - Ct)
    EOL' <- EOL' + LIT(TO, EOL', 0, EOL'.SIZE)
    <Ot, Ct> <- next item in LB
  od
  Oj <- LET(Oj, reverse(EOL'), 0, EOL'.SIZE)
  send <i, Oj, Cj, false> to server through network
  LB <- LB + <Oj, Cj>
  oj <- LIT(oj, HB[k, HB.SIZE])
  perform operation oj on client i.
  HB <- HB + <j, oj, C, false>
    else if (j==i && jflag == true)
      update first entry <j, o', c', false> in HB to <j, o',
          c', true>
      remove first item in LB
      C++
    else
      <k, ok, ck, true> <- entry with smallest k in HB, where
          (j != k && ck > cj)
  oj <- LIT(oj, HB[k, HB.SIZE - 1], 0, HB.SIZE - k )
  undo(HB, Cj)
  perform operation oj on client i.
  for each item <Ot, Ct> in LB
    perform operation Ot = LIT(Ot, HB[Ct, HB.SIZE - 1], 0, HB
       .SIZE - Ct)
    HB <- HB + <i, Ot, Ct, false>
      end
      //redo(HB, oj, Cj)
      C++
    fi
```

Chapter 6

Implementation

In this Chapter, a detailed implementation of the shared task queue application will be presented. Firstly, a general overview of the whole system is presented, including the specific techniques that are used. Then the internal architecture and implementation of each component of the application will be discussed in very detail. Finally the definition of the API and the communication protocol will be presented.

6.1 Distributed Group Task Queue Overview

Shared task queue is a cloud-based service that supports the cooperative work in and between companies. The application is web-based, and can be run whenever a user has access to network and browser. Like traditional web-based applications, the components of the shared task queue application can be divided into three parts, the client side, the server side, and the communication API. The client is responsible for the presentation of a user interface, the generation of the input operations, and the communication with the server. As described in the architecture part, client side consists of view updating component, login component, and the transformation component. The use of the application starts from the login of the user through the client's login component. The changes of the shared information are triggered by clicking buttons or entering information in the client's user interface. The communication with the server is not only deployed at client side but also deployed in the server side, including tasks such as requesting and receiving information from the server. The view updating component in the client side will asynchronously update the views of the client side.

The server part of the application will be responsible for receiving requests from the client side, for sending each received operation to all clients, and

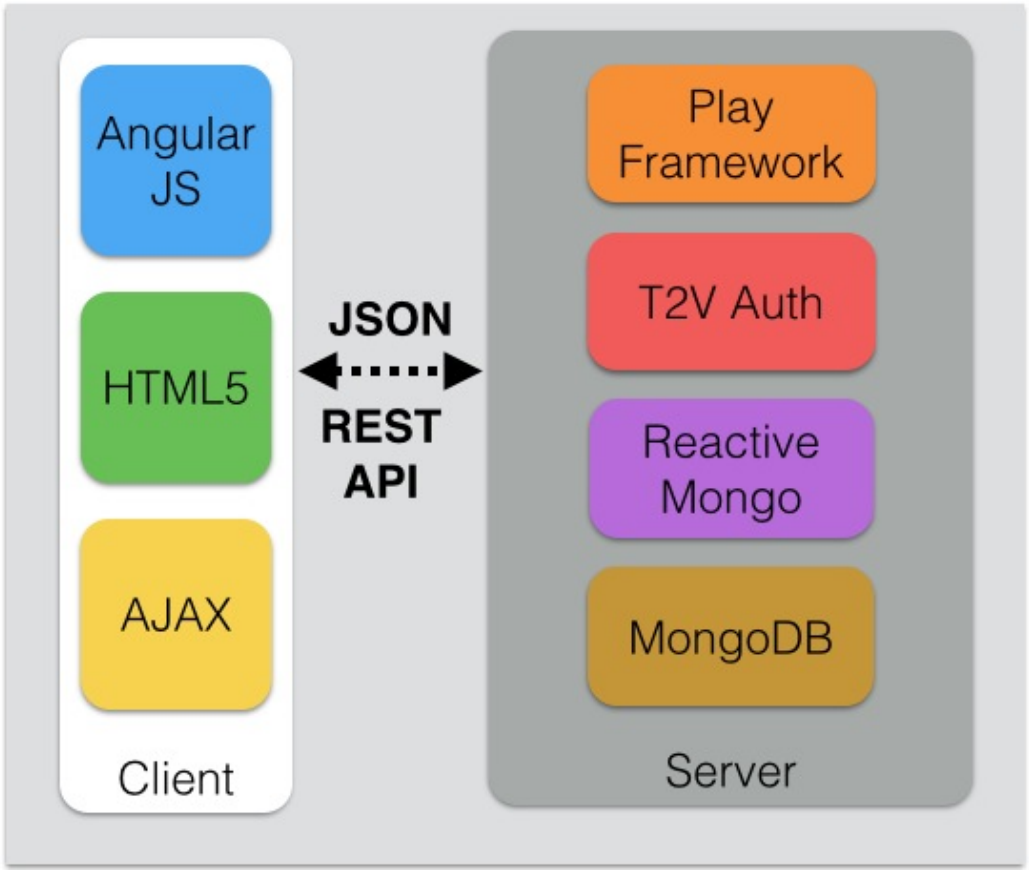

Figure 6.1: Shared Task Queue Overview

saving the task and queues. According to the description in Chapter 4, the server part consists of the login manager component, the notification component, the data update component, and the client list management component. The login management component needs to authorise the user that logs on the system, to update the client lists, and to send initialisation information back to the client. The notification component will notify the clients of a new generated operation and enforce the received operations in a global sequence. The data update component will modify the data stored in the database based on the received operations. For all the data that the application generates, the server will keep them in the data base, including user information, groups and the tasks.

The communication API is the interface between the server and client, to make the interaction between the client and the server to happen smoothly. The API needs to be organised to make the request as clear as possible. The

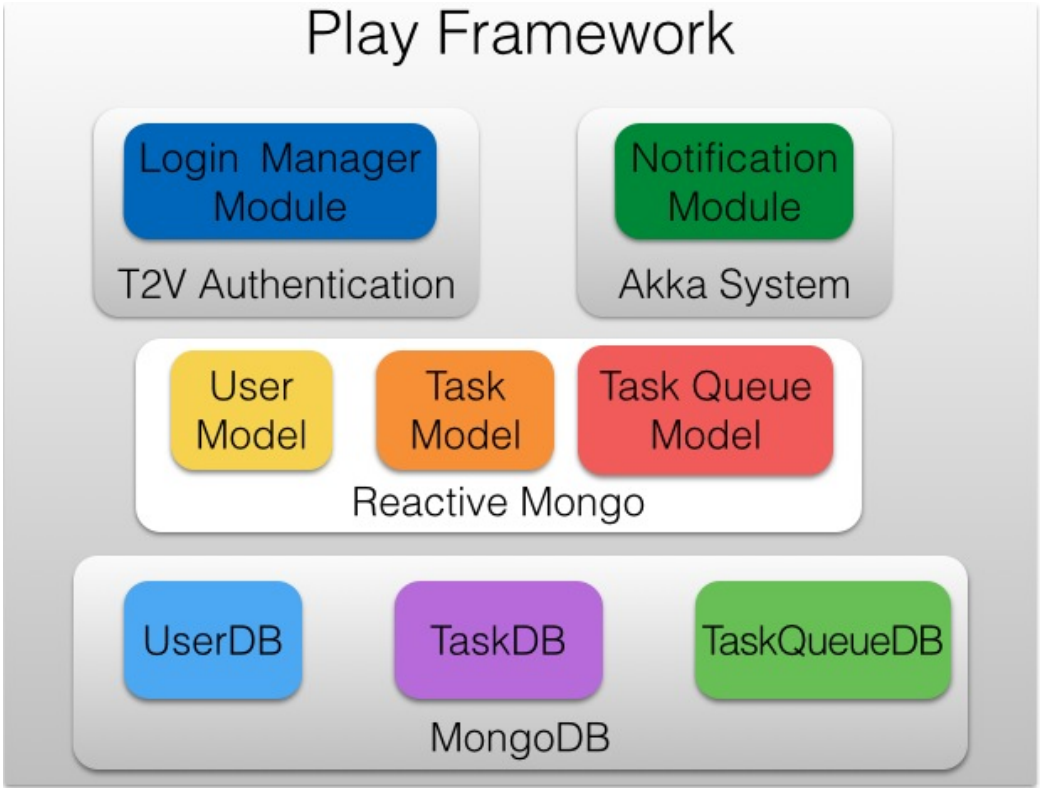

Figure 6.2: Server Side Implementation

data that will be transferred between the clients and the server need to be structured and wrapped in an appropriate format.

6.2 Server Side Component

To accelerate the development process and to add more flexibility to the configuration of the application at the same time, we decided to use open source third-party components. The idea to use open source components also includes considerations of the further usage of the groupware application, and making it capable of being applied to all environments. Figure 6.2 shows how the server components are implemented and the corresponding techniques that we choose.

Play framework is the architecture that we chose for the server part. As described in [28, 31], it is an open source light weight web framework that runs on *JVM*. The light weight feature of this framework can greatly reduce the

configuration job and preparation before development. Also, since it is based on *JVM*, the large amount of supporting components running only on *JVM* is accessible. Furthermore, the framework also supports scala programming language, which is a functional, object oriented programming language, and a better choice for programmers, considering the tedious and complicated way of writing java programs. Play framework uses the stateless architecture to ensure its scalability for later extensions. The inner integrated use of Akka system makes the framework highly reactive to requests and able to generate responses in asynchronous way. The final benefits of using Play framework is its highly active development environment. Developers can have multiple ways to solve the problems they meet during development and the new technologies related third-party components can be integreated easily for Play adopted versions. For example, the T2V authentication component, the authentication module we use in our application for the identifying of a user, has been under actively developing. In this marvellous module, the support for stateless authentication and stateful authentication is available, and to support asynchronous programming, the new versions of the authentication module can authenticate the user asynchronously.

Akka system is the actor model based concurrency control system. As described in [2], Akka system offers a new platform for developing concurrent, fault-tolerant, and highly scalable applications. The development of concurrent programs has been a challenging work for long time, which Akka developers believe is related to wrong tools and the wrong level of abstraction. Using the Actor Model based Akka system, the abstraction level of the concurrent program is raised and the challenges for developing scalable, resilient, and responsive applications decreased. The notification component of the the server side is implemented using the Akka system. Each client will be mapped into one actor and is monitored by the supervisor monitor for its failure and restart. The requests that should be broadcast between each client will be transformed to the corresponding actors that are corresponding to the client. According to the message handling mechanism of Akka system, the message to each actor will be in the same order as the sending sequence of messages, which then means the requests broadcast to each client will be in the same order as they arrived to the server.

MongoDB is an non-SQL database and Reactive Mongo is the driver for MongoDB [9]. The choice of using MongoDB as the database is for the flexibility of the data model. The current data model is designed specificly for the current application. For later extension of the application, the change of the data model is unavoidable. In traditional database systems, this would require an redesign of the data model, but, in MongoDB, old data model can be easily extended to include the new data model. Also, the extension of the

system will have requirements on the scalability of the application, which is a feature of using MongoDB. The reason to select Reactive Mongo is to add asynchronous accessibility to the database. According to the requirements presented on Chapter 3, the asynchronous way of responding is necessary for the application to reduce the response time and increase output of the system, so that an asynchronous database driver is necessary. Reactive Mongo has been popularly used among MongoDB developers and the Play framework version of the Reactive Mongo is adopted.

6.3 Client Side Component

The client side is implemented as a web page application based on the front end techniques. Instead of using static pages, dynamic rendering of the page is chosen, to enable the unblocking operation of the shared task queue. Figure 6.3 shows how the client components are implemented and what techniques the client component use.

The user interface of the client is implemented based on the *Bootstrap toolkits*. The *Bootstrap toolkit* is an open source front end framework developed by Twitter. Bootstrap has abundant components including the layout of the element in a page and a basic javascript action corresponding to specific elements, which greatly reduced the time to configure the presentation of the front end. Also, as bootstrap is well documented, it is quite easy for developers to get started and to adopt the elements for own usage.

The business logic related part of the client side is based on the AngularJS, a structural framework for dynamic web apps. AngularJS enables the developers to use HTML5 as the template and to implement the HTML elements in order to express the application clearly and succinctly. The data binding and dependency injection of Angular reduce the amount of code developers have to type when implementing their applications. The use of AngularJS in our application as the framework for client data rendering reduces the work of the dynamic updating of the views and improves the front end's reactiveness to server pushed responses.

The language used to implement the operational transformation algorithm is Javascript, the declarative language used in front pages. The triggering of execution of the algorithm is based on the input events and the reception of data from the server. The data sent to and received from the server will be handled through ajax polling techniques. The selection of polling will ensure the real-time updating of the data from other remote clients.

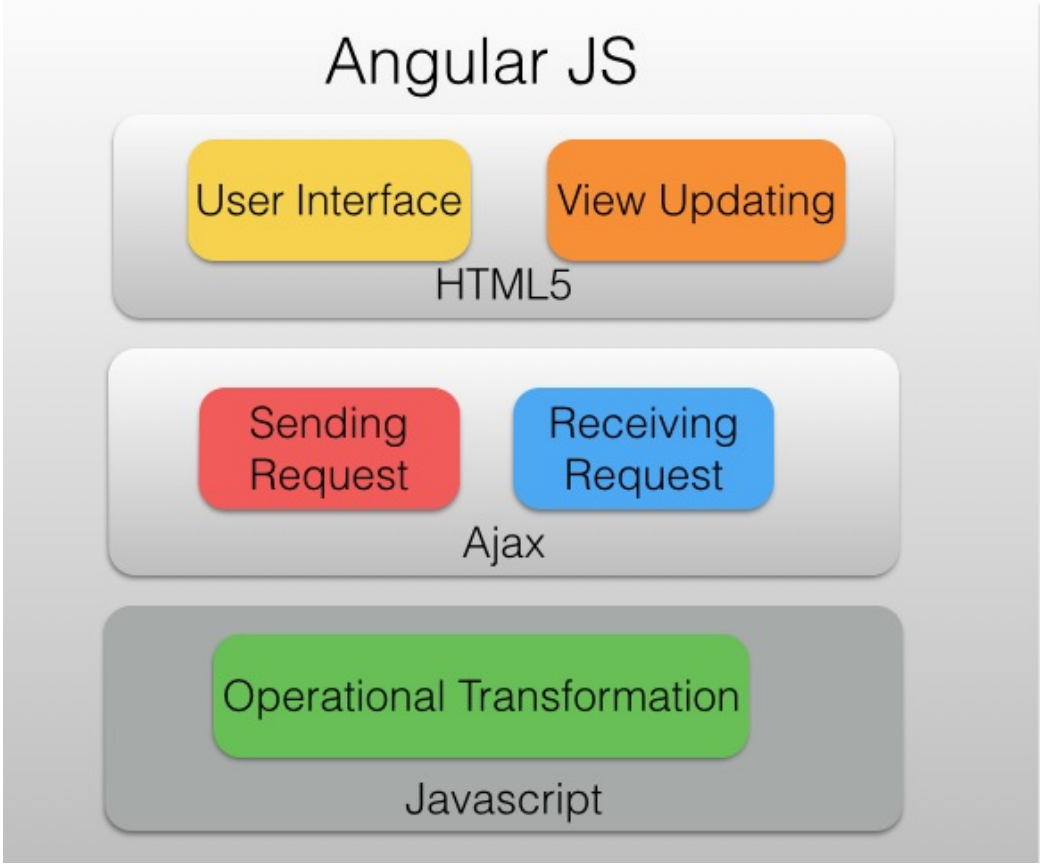

Figure 6.3: Client Side Implementation

6.4 Communication API Component

Communication API will affect the scalability of the application and the stability of the system. The techniques we chose to implement the Communication API is RESTful Architecture and the JSON data format.

RESTful Architecture is important, as described in the requirements of the application in Chapter 3, for the later extension of the system. In our system, the *URLs* are defined strictly under the RESTful requirements. The acquire of data from the server is only be requested through the GET method. *URLs* such as `GET/customer/tasks/unhandled/:id` will be used to get the unprocessed tasks from the server. The sending of data to the server is only done through POST methods. For example, the *URL* `POST/customer/task/newoperation/` will be used to send a new operation to the server. Any updates to data, such as modifying task information, is done by the PUT

methods. The standard of RESTful API simplifies the definition of the URLs and also makes the server APIs easier to be understand.

JSON format is selected as the data structure for data sent between the server and the client. JSON is an expressive and well structured data format and is especially suitable for the transformation of data through networks. Most importantly, both the javascript and scala languages have good support for extracting information from data in JSON format and wrapping the data into the JSON format.

Chapter 7

Testing and Evaluation

In this chapter, we will test the application described in this thesis in laboratory environment. The tools we are going to use for testing, the settings for the tool, and test sets will be discussed. In addition a brief discussion about the result will be presented as the evaluation.

7.1 Concurrency Control Verification

The verification of concurrent applications is different from the verification of traditional program. Concurrent programs have more complex testing activities, since they own unique features such as communication, synchronisation, and nondeterminism [37]. It can be extraordinarily hard to predict in advance where the problems in the various schemes could show up. As a result, even the most clear rules in single thread applications may turn out an unexpected consequence [21]. As concurrent program can result in certain number of interactions between different processes, the number of paths during the execution of an application can be extremely large. This also indicates that the testing of concurrent applications should consider both sequential aspects and asynchronous aspects, which means the methodologies we use in testing traditional applications such as taking a system apart, studying the individual components, and proving the correctness of each divided part will not work [21, 37].

According to [40], there are several challenges in testing a concurrent application. Firstly, it is difficult to perform a static analysis. As concurrent programming always involves several threads and there is nondeterminism between threads, the state graph of the application is extremely complex in most cases. To find problems through a static analysis of the state graph is usually too much work. Hence, an applicable approach for static analysis is

not easy to come up with. Secondly, it is difficult to force a certain path to be executed. In concurrent programming, the existence of nondeterminism could make the execution different even with the same input. When the schedule of processes is decided by the schedule of the system, the forcing of a path to be executed would be almost impossible. The lack of controllability of the application also results in the third challenge in testing concurrent programming, i.e., it is complicated to reproduce a test execution. To test a piece of code, it should be possible for the tester to administer a series of reproducible tests and evaluate the results. The lack of reproducible execution in concurrent programming is beyond the control of a tester.

These challenges push us to find new ways of testing a concurrent program. To describe the behaviour of a distributed system well, verification models are being used , which are the descriptions of high level abstraction of distributed system properties. Once a distributed system is described using verification model, the model can be checked using verification tools. Spin is one of such tools. Spin is a popular open source software for verification of multi-threaded applications. The usages of SPIN can be divided into two categories. The first one is working as a simulator. SPIN can simulate the behaviour of a designed system model as the behaviour in real distributed situations. To help the user to have a more clear and visual view of the execution path of the tested system, SPIN supported a graphical user interface, XSPIN, which has contributed considerable help for developers to analyse the behaviour of a system model for debugging. SPIN supports a high level description language called Promela to model the real system. The emphasis of Promela is on the modelling of synchronisation and coordination of the internal system, instead of computation. So, Promela is a specification language rather than an implementation language. The basic building blocks of SPIN models include asynchronous processes, buffered and unbuffered message channels, synchronising statements, and structured data, which simplifies the process of modelling and verifying the behaviour of coordinative components. However, bare simulation cannot provide any proof of the properties a system is supposed to have. To offer a certificate for certain properties that a system holds, we have to use the second mode of SPIN, i.e., verification. The verification mode of SPIN will simulate all possible execution paths of the modelled system, and tries to find a counter example that violates the properties that the system is supposed to have. The checking method is based on automata. In the verification model, the properties are specified using w-automata and when the designers are verifying the system, any execution path that violates the logic of w-automata properties will be considered as an counter example. SPIN will illustrate the counter-example by creating an execution path using its simulation mode [11, 21].

In our concurrency control testing case, the only property that we need to verify is the consistency of the final content in each distributed client. The simulation part of the concurrency control algorithm in the SPIN model is divided into one server process and several client process. The communication from clients to the server is expressed through the transformation of messages via the channel structure in Promela language. The client process has the OT algorithm implemented and performs each received operation, which is modifying the queue content on the client side. The server process only receives messages from each client and broadcasts the received operations as global operations to each client. To simulate random input in realistic situation, there is an input process corresponding to each client process, which will generate inputs for each client random in both the content and time. For the verification of consistency for the simulated model, the consistency property is defined as following:

P : *The end of input in all clients happened.*

T : *The content queues in all clients are equal.*

$G(P \Rightarrow F(T))$

It should *Globally* hold that whenever P holds, T should *Finally* hold. The condition P is monitored through message passing. When one input process for a client finishes the input task, it will send an end message to the server process. Only when all clients' input end messages have been collected, the server sends a *server end* message to each client process, where the emptiness of the input queue is tested. When the input queue of each client is empty, the system will consider that the end of input in all clients has happened, i.e., P holds. Then, the monitoring of T will start. The monitoring of T is done through checking the queue contents of every client processes and if no difference is found, it means that the consistency holds. Hence, the verification is done. The detailed code of the simulation and verification of Promela model is given in an append of this thesis.

When a model is simulated using the Promela language, the first thing the developers should do is to make sure their model is free from grammatick mistakes. The command of SPIN that can be used to check the grammar is as follows:

```
$ spin OT.pml
```

Using this command line, SPIN will complain about errors in the model if there is any. Besides giving a grammar check for the simulated model, this command line will also provide a random simulation of the model. If there is any output from the model or any printf statement in the model, the result will be seen when the execution terminates, which is helpful in the analysis

of the behaviour of the built model. The output of our model is as showed in figure 7.1

```
→  thesisTest  spin OT_test.pm
              client_0 RECEIVE --> <content: 97, position: 0, global: 0, sender_sequence: 0
                      client_2 RECEIVE --> <content: 107, position: 0, global: 0, sender_sequence: 0
                  client_1 RECEIVE --> <content: 102, position: 0, global: 0, sender_sequence: 0
              client_0 local scaned log_point = 0
              client_0 PERFORM --> <content: 97, position: 0>
                      client_2 local scaned log_point = 0
                      client_2 PERFORM --> <content: 107, position: 0>
                  client_1 local scaned log_point = 0
              client_0 LOG --> <log_index: 0, content: 97, position: 0, global: 0, sequence_count>
                  client_1 PERFORM --> <content: 102, position: 0>
                      client_2 LOG --> <log_index: 0, content: 107, position: 0, global: 0, sequence_count>
              client_0 RECEIVE --> <content: 97, position: 0, global: 1, sender_sequence: 0
                      client_2 RECEIVE --> <content: 97, position: 0, global: 1, sender_sequence: 0
                  client_1 LOG --> <log_index: 0, content: 102, position: 0, global: 0, sequence_count>
                      client_2 global find executed local
              client_0 LOG --> <log_index: 0, content: 97, position: 0, global: 1, sequence_count: 1>
      server ended
                  client_1 RECEIVE --> <content: 97, position: 0, global: 1, sender_sequence: 0
                  client_1 global find executed local
                      client_2 PERFORM --> <content: 97, position: 1>
                      client_2 LOG --> <log_index: 1, content: 97, position: 1, global: 1, sequence_count: 1>
                  client_1 PERFORM --> <content: 97, position: 1>
              client_0 RECEIVE --> <content: 107, position: 0, global: 1, sender_sequence: 0
                  client_1 LOG --> <log_index: 1, content: 97, position: 1, global: 1, sequence_count: 1>
              client_0 global find executed global
                      client_2 RECEIVE --> <content: 107, position: 0, global: 1, sender_sequence: 0
                  client_1 RECEIVE --> <content: 107, position: 0, global: 1, sender_sequence: 0
                      client_2 LOG --> <log_index: 0, content: 107, position: 0, global: 1, sequence_count: 2>
                  client_1 global find executed local
              client_0 PERFORM --> <content: 107, position: 0>
                      client_2 RECEIVE --> <content: 102, position: 0, global: 1, sender_sequence: 0
              client_0 LOG --> <log_index: 1, content: 107, position: 0, global: 1, sequence_count: 2>
                  client_1 global find executed global
                      client_2 global find executed global
              client_0 RECEIVE --> <content: 102, position: 0, global: 1, sender_sequence: 0
              client_0 global find executed global
                  client_1 PERFORM --> <content: 107, position: 1>
                      client_2 global find executed global
                  client_1 LOG --> <log_index: 2, content: 107, position: 1, global: 1, sequence_count: 2>
                  client_1 RECEIVE --> <content: 102, position: 0, global: 1, sender_sequence: 0
              client_0 global find executed global
                      client_2 PERFORM --> <content: 102, position: 0>
              client_0 PERFORM --> <content: 102, position: 0>
                  client_1 LOG --> <log_index: 0, content: 102, position: 0, global: 1, sequence_count: 3>
                      client_2 LOG --> <log_index: 2, content: 102, position: 0, global: 1, sequence_count: 3>
              client_0 LOG --> <log_index: 2, content: 102, position: 0, global: 1, sequence_count: 3>
                      client_2 ended
                  client_1 ended
              client_0 ended
            initial process ended
8 processes created
```

Figure 7.1: Grammar Checking and Random Simulation of SPIN

After making sure that the model is syntactically valid Promela program, we can start to build the verification model. Firstly, we need to produce the source code for a model specific verifier based on the Promela code using SPIN. The command is as following:

```
$ spin -a OT.pml
```

This command line will generate a verifier as a C program. After the execution of this command line, several files with name *pan* but different postfixes will be generated. For example, in our case, the generated verifier files are shown in Figure 7.2

```
→  thesisTest  ls -a pan.*
pan.b pan.c pan.h pan.m pan.p pan.t
→  thesisTest
```

Figure 7.2: Generated Verifier Files

Like normal C programming file, the *pan.h* is the generic header file for the verifier, including variables such as global variables, channels, and process types. File *pan.m* specifies the execution rules for the Promela statements that are contained in the model, and the influences they have on the system state when being successfully performed. File *pan.b* defines the redone policies for statements from file *pan.m* when there is a reversion in the depth-first search. File *pan.t* contains the transition matrix for the built verifier's w-automata. Finally file *pan.c* provides the algorithms for the computation of asynchronous and synchronous products of transitions.

When the verifier is generated, we can compile the verifier to get an executable file for verification. The command line is as follows:

```
$ gcc -o pan pan.c
```

This will generate an executable file named *pan*, which can provide a straight exhaustive verification and the strongest possible verification result when there is sufficient memory to complete the run. The execution of the verification is done using the following command line:

```
$ pan
```

For our Operational Transformation Algorithm, the result is listed in Figure 7.3.

There are 8 processes created for the our model, 3 clients, 3 input processes for clients, one server process, and an initial process. As a result of the limited memory in the running machine, there are only two input elements generated by each input process. However, the puzzle shown in Figure 5.1 is not even as complicated as in this limited case. As we can see from the result figure, no violation of assertion was reported after searching all states. Hence, we can say that our concurrency control algorithm is correct in maintaining the consistency between distributed clients in these limited yet complicated enough situations.

```
→  thesisTest  ./pan
hint: this search is more efficient if pan.c is compiled -DSAFETY

(Spin Version 6.4.3 -- 16 December 2014)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 4064 byte, depth reached 241, errors: 0
   129726 states, stored
    63317 states, matched
   193043 transitions (= stored+matched)
    35177 atomic steps
hash conflicts:        83 (resolved)

Stats on memory usage (in Megabytes):
  506.247       equivalent memory usage for states (stored*(State-vector + overhead))
  496.876       actual memory usage for states (compression: 98.15%)
                state-vector as stored = 3988 byte + 28 byte overhead
  128.000       memory used for hash table (-w24)
    0.534       memory used for DFS stack (-m10000)
    1.320       memory lost to fragmentation
  624.094       total actual memory usage


unreached in proctype client_process
        OT_test.pm:235, state 10, "(1)"
        OT_test.pm:252, state 21, "(1)"
        OT_test.pm:281, state 55, "(1)"
        OT_test.pm:184, state 86, "(1)"
        OT_test.pm:297, state 94, "(1)"
        OT_test.pm:311, state 107, "(1)"
        OT_test.pm:360, state 144, "(1)"
        OT_test.pm:76, state 155, "local_op.position = (local_op.position+1)"
        OT_test.pm:79, state 158, "local_op.position = (local_op.position+1)"
        OT_test.pm:83, state 163, "result[client_id]!local_op.name,local_op.position,local_op.content"
        OT_test.pm:150, state 167, "(((op_log[eto_index].op.name==insert)&&(local_op.name==insert)))"
        OT_test.pm:373, state 169, "result[client_id]?local_op.name,local_op.position,local_op.content"
        OT_test.pm:387, state 197, "(1)"
        OT_test.pm:76, state 208, "local_op.position = (local_op.position+1)"
        OT_test.pm:79, state 211, "local_op.position = (local_op.position+1)"
        OT_test.pm:83, state 216, "result[client_id]!local_op.name,local_op.position,local_op.content"
        OT_test.pm:150, state 220, "(((op_log[eto_index].op.name==insert)&&(local_op.name==insert)))"
        OT_test.pm:399, state 222, "result[client_id]?local_op.name,local_op.position,local_op.content"
        OT_test.pm:76, state 227, "local_op.position = (local_op.position+1)"
        OT_test.pm:79, state 230, "local_op.position = (local_op.position+1)"
        OT_test.pm:83, state 235, "result[client_id]!local_op.name,local_op.position,local_op.content"
        OT_test.pm:150, state 239, "(((op_log[eto_index].op.name==insert)&&(local_op.name==insert)))"
        OT_test.pm:404, state 241, "result[client_id]?local_op.name,local_op.position,local_op.content"
        OT_test.pm:184, state 301, "(1)"
        OT_test.pm:429, state 309, "(1)"
        OT_test.pm:447, state 323, "(1)"
        (26 of 349 states)
unreached in proctype generate_process
        (0 of 19 states)
unreached in proctype server_process
        (0 of 34 states)
unreached in proctype init_process
        (0 of 42 states)

pan: elapsed time 3.57 seconds
pan: rate 36337.815 states/second
```

Figure 7.3: Verify Results

Chapter 8

Conclusions

In this chapter we conclude this thesis. Some experiences we got from designing the group shared task queue application will be addressed and the contribution of this thesis will be concluded.

8.1 Conclusions

Collaboration has become increasingly important in our life, especially in our working environment. As computers have been so successful in supporting individual work, it is very natural that researchers seek for ways of using computers in collaborative work, i.e., the collaborative software or groupware. Though the number of researchers in groupware has been increasing over the years, the definition of groupware remains unclear. However, in all those various definitions about groupware, the keyword "collaboration" is commonly seen. The support for collaboration adds extra challenges to the design and implementation of groupware. Groupware has general requirements, such as appropriate user interaction handling, coordination, support for distribution, visualisation consistency, and data hiding. All these requirements can be the basis of evaluation for judging the success of particular groupware. Compared to single-user applications, the design and implementation of groupware are more challenging. The difficulties of designing a groupware include a deep understanding of task requirements, appropriate methods for handling group member awareness, and offering some flexibility to groupware application. Nowadays, there are two approaches toward the building of groupware, the collaboration transparency and collaboration awareness approaches. The former approach will build the groupware under the framework of the corresponding single-user application, taking use of its user interface, business logic, part of the source code, and so on. The

latter approach will build an absolutely new application based on the roles of each participant in the collaborative task. No matter what approach we choose, we must consider the architecture of building the application carefully. There are two widely used architectures in groupware design. The first one is the centralised architecture, which is very popular due to its easy implementation and convenient maintenance features. However, this approach will deploy too much burden to the central server, and the blocking feature of the user interface will lead to bad user experience. The second one is the replicated architecture. Replicated architecture will ensure the good performance of the whole system and nice user experience, but is far more difficult to be implemented by the developer.

In this thesis, we presented a software solution for a shared task queue, which is a component in a real software project. The shared task queue will allow multiple users to operate on the content in the queue in parallel. To make this component meet the requirements of the project and also to be as reusable as possible, we explored the requirements that have to be taken into consideration. To make the component to integrate to the whole project, we added the RESTful API requirement. For the future extensions of the project, we decided that the component should be scalable and highly available. Also, to extend the use scenarios of the groupware component, we found out that the support for offline operations and late joining of the users is necessary. Finally, to give a better user experience, we also found out that non-blocking IO and real-time cooperation support are necessary.

This thesis design the architecture of the shared task queue groupware through two steps. To select a better architecture for the shared task queue component, we firstly gave the design for two traditional architectures that are adopted to our specific use case. The analysis of the two architectures are given, including the benefits of each architecture and the challenges or potential problems in each architecture. The analysis of the two architectures showed that in these two architectures, one's cons turned out to be the pros in the other, and vice versa. This led to the idea of combining these two architectures together and coming up with our own architecture for the groupware component, i.e., the second step. The combined architecture consists of two parts, the client component and the central server component. The central server is responsible for receiving requests from clients, giving a sequence number to that request, and sending this request to all other clients. The client component will take care of the inputs from the user interface, receive the response from the server, solve the local conflicts based on the priority of each operation, and display the result of the operations on the screen. The combined architecture has a central server for the assignment of a global sequence but distributed the conflicts solving part to each client.

Hence, the architecture in this thesis has no bottlenecks on the server side but the clients are not too complicated to implement at the same time. The essential part for the combined architecture to run is the conflicts solving algorithm.

To address the concurrency problem, we adopted the conflict solving algorithm from the OT algorithm, which is an algorithm used for solving conflicts in groupware. The adoption of the algorithm starts with defining the transformation model, a matrix used for transforming an operation into another operation when certain operation happened before this operation. The main idea of the algorithm is to put every operation into the queue, before it is showed on the screen, including the auxiliary information such as local or global priority. Then based on the priority of an operation that is going to be executed, the algorithm will pick out the operations in the executed list with lower priority compared to the executing operation. Then the executing operation will be transformed using the operation picked out and will get executed. The benefits of this algorithm are that it will not block the client component when resolving the conflicts and also the history list of the operations can be reduced greatly.

Then we implemented the components in the architecture design as a web application under Play! Framework. The server side uses the Akka system to implement the notification of requests and to solve the data consistency problem during the login and logout of users. For totally asynchronous features of the whole system, MongoDB database and Reactive Mongo were selected for storing and accessing the data. In the client side, Javascript has been chosen as the language for implementing the client logic. AngularJS has been selected as the framework for client code organisation. The communication between the client side and the server side is done by Ajax calling and a polling mechanism is implemented in the client side for the real-time data updating. Finally we verified the implementation of the algorithm using the SPIN Model Checker.

The result of the thesis shows that a combined replicated architecture is a better architecture than centralised architecture in the shared task queue groupware. The adoption of OT algorithm could solve the concurrency control problem in shared task queue groupware. The testing of OT algorithm using SPIN model checker has opened the door for verifying the correctness of adopted OT algorithm using simulation tools.

8.2 Future Work

There are several possibilities for future work regarding to the work presented in this Thesis.

First of all, the OT algorithm can run more smoothly if there is a garbage collection scheme on the client side. As the number of operations generated by clients increasing, the buffer size of HB will increase. Hence, inefficient memory consumption exists. Also, as the size of HB increases, the time for searching operations in HB will raise. If a garbage collection scheme that deleting all those executed operations from HB is implemented on client side, the size of HB can be kept at a relatively small size. As a result, the algorithm could run in a more efficient way.

In addition, the verification of adopted OT algorithm is only conducted under limited inputs. For verifying more complicated cases, the model simulated using Promela should be optimised.

Bibliography

[1] Allen., C. Definitions of groupware, Fall 1990.

[2] Allen, J. *Effective Akka*. O'Reilly Media, Inc., 2013.

[3] Begole, J., Struble, C., and Shaffer, C. Leveraging java applets: toward collaboration transparency in java. *Internet Computing, IEEE 1*, 2 (Mar 1997), 57–64.

[4] Belqasmi, F., Glitho, R., and Fu, C. Restful web services for service provisioning in next-generation networks: a survey. *Communications Magazine, IEEE 49*, 12 (2011), 66–73.

[5] Brooks, F. *No silver bullet*. April, 1987.

[6] Cardellini, V., Colajanni, M., and Philip, S. Y. Dynamic load balancing on web-server systems. *IEEE Internet computing*, 3 (1999), 28–39.

[7] Carstensen, P. H., and Schmidt, K. Computer supported cooperative work: New challenges to systems design. In *In K. Itoh (Ed.), Handbook of Human Factors* (1999), Citeseer.

[8] Chaffey, D. *GroupWare, Workflow and Intranets: Reengineering the Enterprise with Collaborative Software*. Butterworth-Heinemann, Newton, MA, USA, 1998.

[9] Chodorow, K. *MongoDB: the definitive guide*. O'Reilly Media, Inc., 2013.

[10] Coleman, D. Groupware. In *Groupware: technology and applications*, D. Coleman and R. Khanna, Eds. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995, ch. Groupware Technology and Applications: An Overview of Groupware, pp. 3–41.

[11] Spin Main Page. `http://spinroot.com/spin/whatispin.html`, 2006.

[12] Collaborative software. `https://en.wikipedia.org/w/index.php?title=Collaborative_software&oldid=674752948`, 2015.

[13] Representational state transfer. `https://en.wikipedia.org/w/index.php?title=Representational_state_transfer&oldid=680518039`, 2015.

[14] Dabek, F., Zeldovich, N., Kaashoek, F., Mazières, D., and Morris, R. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop* (2002), ACM, pp. 186–189.

[15] Ellis, C. A., and Gibbs, S. J. Concurrency control in groupware systems. *SIGMOD Rec. 18*, 2 (June 1989), 399–407.

[16] Ellis, C. A., Gibbs, S. J., and Rein, G. Groupware: Some issues and experiences. *Commun. ACM 34*, 1 (Jan. 1991), 39–58.

[17] Ensor, B. How can we make groupware practical? (panel). In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 1990), CHI '90, ACM, pp. 87–89.

[18] Greenberg, S., and Marwood, D. Real time groupware as a distributed system: Concurrency control and its effect on the interface. In *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work* (New York, NY, USA, 1994), CSCW '94, ACM, pp. 207–217.

[19] Grudin, J. Computer-supported cooperative work: History and focus. *Computer 27*, 5 (May 1994), 19–26.

[20] Gutwin, C. A., Lippold, M., and Graham, T. C. N. Real-time groupware in the browser: Testing the performance of web-based networking. In *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work* (New York, NY, USA, 2011), CSCW '11, ACM, pp. 167–176.

[21] Holzmann, G. *Spin Model Checker, the: Primer and Reference Manual*, first ed. Addison-Wesley Professional, 2003.

[22] Hughes, J., Randall, D., and Shapiro, D. Cscw: Discipline or paradigm? a sociological perspective. In *Proceedings of the Second Conference on European Conference on Computer-Supported Cooperative Work* (Norwell, MA, USA, 1991), ECSCW'91, Kluwer Academic Publishers, pp. 309–323.

[23] Johansen, R. *GroupWare: Computer Support for Business Teams*. The Free Press, New York, NY, USA, 1988.

[24] Johnson-Lenz, P., Johnson-Lenz, T., and of Technology. Computerized Conferencing & Communications Center, N. J. I. *The Evolution of a Tailored Communications Structure: The TOPICS System*. New Jersey Institute of Technology. Computerized Conferencing and Communications Center. Research report. Computer & Information Science Department, New Jersey Institute of Technology, 1981.

[25] Jorgenson, D. W., and Vu, K. Information technology and the world economy. *Scandinavian Journal of Economics 107*, 4 (2005), 631–650.

[26] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.

[27] Lauwers, J. C., and Lantz, K. A. Collaboration awareness in support of collaboration transparency: Requirements for the next generation of shared window systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 1990), CHI '90, ACM, pp. 303–311.

[28] Layka, V. Play with java and scala. In *Learn Java for Web Development*. Springer, 2014, pp. 355–382.

[29] Li, D., and Li, R. Transparent sharing and interoperation of heterogeneous single-user applications. In *Proceedings of the 2002 ACM Conference on Computer Supported Cooperative Work* (New York, NY, USA, 2002), CSCW '02, ACM, pp. 246–255.

[30] Nunamaker, Jr., J. F., Briggs, R. O., and Mittleman, D. D. Groupware. In *Groupware: technology and applications*, D. Coleman and R. Khanna, Eds. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995, ch. Electronic Meeting Systems: Ten Years of Lessons Learned, pp. 149–193.

[31] Reelsen, A. *Play Framework Cookbook*. Packet Publishing Ltd, 2011.

[32] Reinhard, W., Schweitzer, J., Volksen, G., and Weber, M. Cscw tools: concepts and architectures. *Computer 27*, 5 (May 1994), 28–36.

[33] Rissanen, H.-M., Mecklin, T., and Opsenica, M. Design and implementation of a restful ims api. In *Wireless and Mobile Communications (ICWMC), 2010 6th International Conference on* (2010), IEEE, pp. 86–91.

[34] Rittel, H. W., and Webber, M. M. Dilemmas in a general theory of planning. *Policy sciences 4*, 2 (1973), 155–169.

[35] Schmidt, K., and Bannon, L. Taking cscw seriously. *Computer Supported Cooperative Work (CSCW) 1*, 1-2 (1992), 7–40.

[36] Shaw, M., and Garlan, D. *Software architecture: perspectives on an emerging discipline*, vol. 1. Prentice Hall Englewood Cliffs, 1996.

[37] Souza, S. R. S., Brito, M. A. S., Silva, R. A., Souza, P. S. L., and Zaluska, E. Research in concurrent software testing: A systematic review. In *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging* (New York, NY, USA, 2011), PADTAD '11, ACM, pp. 1–5.

[38] Sun, C., Jia, X., Zhang, Y., Yang, Y., and Chen, D. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact. 5*, 1 (Mar. 1998), 63–108.

[39] Tanenbaum, A. S., and Van Steen, M. *Distributed systems*. Prentice-Hall, 2007.

[40] Yang, C.-S. D., and Pollock, L. L. All-uses testing of shared memory parallel programs.

[41] Zuravleff, W. K., Semmelmeyer, M., Robinson, T., and Furman, S. Non-blocking load buffer and a multiple-priority memory system for real-time multiprocessing, Sept. 22 1998. US Patent 5,812,799.

Appendix A

Promela Code for Evaluation

```
mtype = { insert, delete, modify, empty_op, end_op}
typedef Operation {
  mtype name
  int position
  byte content
};

typedef Q_Node {
  int client_id
  Operation op
  int sequence_count
  bool global
};

typedef Perform_Param {
  Operation op
  int length
};

typedef Content {
  byte c[20];
}

int DEBUG = 1;

int client_num = 3;

//the content of the queue
Content content[client_num];

//channel for sending request to server
chan server_request = [20] of {Q_Node};
```

```
//channels for sending response to clients
chan response[client_num] = [20] of {Q_Node};

//channels for storing the return values of transform
chan result[client_num] = [1] of {Operation};

//channels for operations waiting for transforming
chan op_queue[client_num] = [20] of {Q_Node}

//channels for ensuring the reading of sequence number
chan sequence[client_num] = [1] of {int}

//channels for controlling the input taken one by one
chan input_mutex[client_num] = [1] of {int}

//channels for end of inputs
chan input_end[client_num] = [1] of {int}

//channels for end of clients processing
chan client_end[client_num] = [1] of {int}

//channels for end of server processing
chan server_end = [1] of {int}

//channels for sending input number
chan input_total = [1] of {int}

//define the transformation matrix;
//prior means o2 occured before o1;
inline insert_modify(client_id1, o1, o2, prior) {
  if
    :: (o1.position <= o2.position) -> {
      o2.position = o2.position + 1;
    }
    :: else -> skip
  fi
  result[client_id] ! o2
}

inline insert_insert(client_id, o1, o2, prior) {
  if
    :: (o1.position < o2.position) -> {
      o2.position = o2.position + 1;
    }
    :: (o1.position == o2.position && prior != 1) -> {
      o2.position++;
    }
    :: else -> skip
  fi
```

```
  result[client_id] ! o2
}

inline insert_delete(client_id, o1, o2, prior) {
  if
    :: (o1.position <= o2.position) -> {
      o2.position = o2.position + 1;
    }
    :: else -> skip
  fi
  result[client_id] ! o2
}

inline delete_modify(client_id, o1, o2, prior) {
  if
    :: (o1.position < o2.position) ->
      o2.position = o2.position - 1;
    :: (o1.position == o2.position) ->
      o2.name = empty_op
    :: else -> skip
  fi
  result[client_id] ! o2
}

inline delete_insert(client_id, o1, o2, prior) {
  if
    :: (o1.position < o2.position) -> {
      o2.position = o2.position - 1;
    }
    :: else -> skip
  fi
  result[client_id] ! o2
}

inline delete_delete(client_id, o1, o2, prior) {
  if
    :: (o1.position < o2.position) -> {
      o2.position = o2.position - 1;
    }
    :: (o1.position == o2.position) -> {
      o2.name = empty_op
    }
    :: else -> skip
  fi
  result[client_id] ! o2
}

inline modify_modify(client_id, o1, o2, prior) {
  if
```

```
    :: (o1.position == o2.position && prior == 1) ->
      o2.name = empty_op
    :: else -> skip
  fi
  result[client_id] ! o2
}

inline modify_insert(client_id, o1, o2, prior) {
  result[client_id] ! o2
}

inline modify_delete(client_id, o1, o2, prior) {
  result[client_id] ! o2
}

inline transformOperation(client_id, o1, o2, prior) {
  if
  :: (o1.name == insert && o2.name == modify) ->
     insert_modify(client_id, o1, o2, prior)
  :: (o1.name == insert && o2.name == insert) ->
     insert_insert(client_id, o1, o2, prior)
  :: (o1.name == insert && o2.name == delete) ->
     insert_delete(client_id, o1, o2, prior)
  :: (o1.name == modify && o2.name == modify) ->
     modify_modify(client_id, o1, o2, prior)
  :: (o1.name == modify && o2.name == insert) ->
     modify_insert(client_id, o1, o2, prior)
  :: (o1.name == modify && o2.name == delete) ->
     modify_delete(client_id, o1, o2, prior)
  :: (o1.name == delete && o2.name == modify) ->
     delete_modify(client_id, o1, o2, prior)
  :: (o1.name == delete && o2.name == insert) ->
     delete_insert(client_id, o1, o2, prior)
  :: (o1.name == delete && o2.name == delete) ->
     delete_delete(client_id, o1, o2, prior)
  fi
}

inline performOperation(client_id, pf_param) {
  atomic {

  if
  :: (DEBUG == 1) -> printf("client_%d PERFORM --> <content:
     %d, position: %d>\n", client_id, pf_param.op.content,
     pf_param.op.position);
  :: else -> skip
  fi
  if
  :: (pf_param.op.name == insert) -> {
```

```
    int initial = pf_param.op.position
    byte pre = content[client_id].c[initial]
    byte post = content[client_id].c[initial + 1]
    content[client_id].c[initial] = pf_param.op.content;
    do
    :: (initial < pf_param.length) ->
      post = content[client_id].c[initial + 1]
      content[client_id].c[initial + 1] = pre
      pre = post
      initial = initial + 1;
    :: else ->
      break
    od
    }
  :: (pf_param.op.name == delete) -> {
    int initial = pf_param.op.position
    byte post = content[client_id].c[initial + 1]
    do
    :: (initial < length - 1) ->
      post = content[client_id].c[initial + 1]
      content[client_id].c[initial] = post
      initial = initial + 1;
    :: else ->
      break
    od
    }
  :: (pf_param.op.name == modify) -> {
    content[client_id].c[pf_param.op.position] = pf_param.op.
       content;
    }
  :: else -> skip
  fi
  }
}

typedef InsertContent {
  byte c [5];
}

InsertContent ic[client_num];

proctype client_process(int client_id) {

  Q_Node op_log[20]
  int head = 0;
  int tail = 0;
  int log_count = 0;
  int bool_full = 0;
  int global;
```

```
int id;
int sequence_count = 0;
int length = 0;
Operation op;
Q_Node rec_op;

do
:: op_queue[client_id] ? rec_op ->  {
  id = rec_op.client_id
  //sequence_count = rec_op.sequence_count
  op.name = rec_op.op.name;
  op.position = rec_op.op.position;
  op.content = rec_op.op.content;
  global = rec_op.global

  if
  :: (DEBUG == 1) -> printf("client_%d RECEIVE --> <content
     : %d, position: %d, global: %d, sender_sequence: %d\n
     ", client_id, op.content, op.position, global, rec_op.
     sequence_count);
  :: else -> skip
  fi


  int log_point = 0;
  int eto_index;
  Operation local_op
  if
  :: (id == client_id && global == 0) -> {
    do
    :: (log_point < log_count &&
      op_log[log_point].client_id != client_id &&
      op_log[log_point].global == 1 &&
      op_log[log_point].sequence_count > rec_op.
         sequence_count) ->

      if
      :: (DEBUG == 1) -> printf("client_%d find executed
         global\n", client_id);
      :: else -> skip
      fi

      eto_index = log_point + 1;
      result[client_id] ! op_log[log_point].op
      result[client_id] ? local_op

      do
        :: (eto_index < log_count && op_log[eto_index].
           sequence_count <=) ->
```

```
        transformOperation(client_id, op_log[eto_index].
           op, local_op, 0)
        eto_index++
        result[client_id] ? local_op

      :: else -> break
    od

    transformOperation(client_id, local_op, op, 1);
    result[client_id] ? op
    log_point = log_point + 1
  :: (log_point < log_count &&
    (op_log[log_point].client_id == id ||
    op_log[log_point].global != 1 ||
    op_log[log_point].sequence_count <= rec_op.
       sequence_count)) ->
    log_point = log_point + 1
  :: else -> break;
  od

  if
    :: (DEBUG == 1) -> printf("client_%d local scaned
       log_point = %d\n", client_id, log_point);
    :: else -> skip
  fi



  Perform_Param pf_param;
  pf_param.op.name = op.name;
  pf_param.op.position = op.position;
  pf_param.op.content = op.content;

  pf_param.length = length;

  performOperation(client_id, pf_param);

  if
  :: (op.name == insert) -> length = length + 1
  :: else -> skip
  fi
  op_log[log_count].client_id = id
  op_log[log_count].sequence_count = sequence_count

  op_log[log_count].op.name = op.name
  op_log[log_count].op.position = op.position
  op_log[log_count].op.content = op.content

  op_log[log_count].global = global
```

```
    server_request ! op_log[log_count];

    if
      :: (DEBUG == 1) -> printf("client_%d LOG --> <
         log_index: %d, content: %d, position: %d, global:
         %d, sequence_count>\n", client_id, log_count,
         op_log[log_count].op.content, op_log[log_count].op
         .position, op_log[log_count].global, op_log[
         log_count].sequence_count);
      :: else -> skip
    fi

    log_count++;
    input_mutex[client_id] ! 1;
  }
  :: ((id == client_id) && (global == 1)) -> {
    int index = 0;
    do
    :: (index < log_count &&
      op_log[index].client_id == id &&
      op_log[index].global == 0) ->
      //do not need to judge name and content here, since
         the first sented
      //local input will get the global reply first.

      op_log[index].op.name = op.name
      op_log[index].op.content = op.content
      op_log[index].global = 1
      break
    :: (index < log_count &&
      (op_log[index].client_id != id ||
      op_log[index].global != 0 )) ->
      index = index + 1
    :: else -> break;
    od

    sequence[client_id] ? sequence_count;
    sequence_count = sequence_count + 1
    op_log[index].sequence_count = sequence_count;
    sequence[client_id] ! sequence_count;

    if
      :: (DEBUG == 1) -> printf("client_%d LOG --> <
         log_index: %d, content: %d, position: %d, global:
         %d, sequence_count: %d>\n", client_id, index,
         op_log[index].op.content, op_log[index].op.
         position, op_log[index].global, op_log[index].
         sequence_count);
      :: else -> skip
```

```
    fi


  }
  :: ((id != client_id) && (global == 1)) -> {

    int found = 0;
    do
    :: (log_point < log_count &&
      op_log[log_point].client_id != id &&
      op_log[log_point].global == 1 &&
      op_log[log_point].sequence_count > rec_op.
         sequence_count) ->

      if
      :: (DEBUG == 1) -> printf("client_%d global find
         executed global\n", client_id);
      :: else -> skip
      fi


        eto_index = log_point + 1;
      local_op.name = op_log[log_point].op.name
      local_op.position = op_log[log_point].op.position
      local_op.content = op_log[log_point].op.content
      do
        // global operations has lower priority than all
           later operations
        :: (eto_index < log_count) ->
          transformOperation(client_id, op_log[eto_index].
             op, local_op, 0)
          eto_index++
          result[client_id] ? local_op

        :: else -> break
      od

      transformOperation(client_id, local_op, op, 1) //op
         is global and local_op is global
      result[client_id] ? op

      log_point = log_point + 1
    :: (log_point < log_count &&
      op_log[log_point].global == 0) ->

      if
      :: (DEBUG == 1) -> printf("client_%d global find
         executed local\n", client_id);
      :: else -> skip
```

```
  fi

  eto_index = log_point + 1;

  local_op.name = op_log[log_point].op.name
  local_op.position = op_log[log_point].op.position
  local_op.content = op_log[log_point].op.content
  do
    :: (eto_index < log_count && op_log[eto_index].
       global == 0) ->
      transformOperation(client_id, op_log[eto_index].
         op, local_op, 0)
      eto_index++
      result[client_id] ? local_op

    :: (eto_index < log_count && op_log[eto_index].
       global == 1) ->
      transformOperation(client_id, op_log[eto_index].
         op, local_op, 1)
      eto_index++
      result[client_id] ? local_op
    :: else -> break
  od

  //op is global and local_op is local
  transformOperation(client_id, local_op, op, 0)
  result[client_id] ? op
  log_point = log_point + 1
:: (log_point < log_count && (op_log[log_point].
   client_id == id ||
  (op_log[log_point].sequence_count <= rec_op.
     sequence_count &&
  op_log[log_point].global == 1))) ->
  log_point = log_point + 1
:: else -> break;
od

sequence[client_id] ? sequence_count;
//PERFORM OP
Perform_Param pf_param;
pf_param.op.name = op.name;
pf_param.op.position = op.position;
pf_param.op.content = op.content;
pf_param.length = length;
performOperation(client_id, pf_param);
if
:: (op.name == insert) -> length = length + 1
:: else -> skip
fi
```

```
      sequence_count = sequence_count + 1
      sequence[client_id] ! sequence_count;

      //UPDATE LOG QUEUE;
      op_log[log_count].client_id = id
      op_log[log_count].sequence_count = sequence_count

      op_log[log_count].op.name = op.name
      op_log[log_count].op.position = op.position
      op_log[log_count].op.content = op.content

      op_log[log_count].global = global

      if
        :: (DEBUG == 1) -> printf("client_%d LOG --> <
           log_index: %d, content: %d, position: %d, global:
           %d, sequence_count: %d>\n", client_id, log_count,
           op_log[log_count].op.content, op_log[log_count].op
           .position, op_log[log_count].global, op_log[
           log_count].sequence_count)
        :: else -> skip
      fi

      log_count++;
    }
    fi

  }

  :: nempty(server_end) -> atomic {
    if
      :: empty(op_queue[client_id]) ->
        client_end[client_id] ! 1
        goto end
      :: nempty(op_queue[client_id]) ->
        skip
    fi
    }
  od

end:
  if
    :: (DEBUG == 1) -> printf("client_%d ended \n", client_id
       );
    :: else -> skip
  fi

}
```

```
proctype generate_process(int client_id) {
  Q_Node rec_op
  int position = 0;
  int approval = 0;
  do
  :: (position < 1) ->

    input_mutex[client_id] ? approval;
    rec_op.client_id = client_id;
    sequence[client_id] ? rec_op.sequence_count;
    sequence[client_id] ! rec_op.sequence_count;
    rec_op.op.name = insert;
    rec_op.op.position =  position;
    rec_op.op.content = ic[client_id].c[position];
    rec_op.global = 0;
    position++;
    op_queue[client_id] ! rec_op;
  :: else -> break
  od
  input_end[client_id] ! 1;
end:
}

active proctype server_process () {
  Q_Node op_node;
  int input_num = 0;
  int input_count = 0;
  int control = 0;
  do
  :: server_request ? op_node -> atomic {
    op_node.global = 1;

    int i = 0;
    do
      :: (i < client_num) ->
        op_queue[i] ! op_node;
        i++
      :: else -> break;
    od
    input_count++;
    control = 1;
    }
  :: (control == 1 && input_total ? [input_num]) ->

    input_total ? input_num
    if
      :: (input_num == input_count) ->
        server_end ! 1
```

```
        goto end
      :: else ->
        input_total ! input_num
        control = 0;
        //skip
    fi
  od

end:
  if
    :: (DEBUG == 1) -> printf("server ended\n");
    :: else -> skip
  fi
}

active proctype init_process () {

  int client1, client2, client3;
  client1 = 0;
  client2 = 1;
  client3 = 2;

  ic[client1].c[0] = 'a';
  ic[client1].c[1] = 'b';
  ic[client1].c[2] = 'c';
  ic[client1].c[3] = 'd';
  //ic[client1].c[4] = 'e';

  ic[client2].c[0] = 'f';
  ic[client2].c[1] = 'g';
  ic[client2].c[2] = 'h';
  ic[client2].c[3] = 'i';
  ic[client2].c[4] = 'j';

  ic[client3].c[0] = 'k';
  ic[client3].c[1] = 'l';
  ic[client3].c[2] = 'm';
  ic[client3].c[3] = 'n';
  ic[client3].c[4] = 'o';

  sequence[client1] ! 0;
  sequence[client2] ! 0;
  sequence[client3] ! 0;

  input_mutex[client1] ! 1;
  input_mutex[client2] ! 1;
  input_mutex[client3] ! 1;

  run client_process(client1)
```

```
  run client_process(client2)
  run client_process(client3)

  run generate_process(client1)
  run generate_process(client2)
  run generate_process(client3)

  int content_index = 0

  int input_num1 = 0;
  int input_num2 = 0;
  int input_num3 = 0;

  input_end[client1] ? input_num1;
  input_end[client2] ? input_num2;
  input_end[client3] ? input_num3;

  input_total ! (input_num1 + input_num2)

  client_end[client1] ? _
  client_end[client2] ? _
  client_end[client3] ? _

end:  do
      :: (content_index < 10) ->
        assert(content[client1].c[content_index] == content[
           client2].c[content_index])
        assert(content[client1].c[content_index] == content[
           client2].c[content_index] &&
          content[client1].c[content_index] == content[
             client3].c[content_index])
        content_index ++
      :: else -> break;
    od

    if
      :: (DEBUG == 1) -> printf("initial process ended\n");
      :: else -> skip
    fi
}
```